RAPID PROTOTYPING AND EXPLORATION ENVIRONMENT FOR GENERATING C-TO-HARDWARE-COMPILERS

EINGEREICHTE DISSERTATION
VORGELEGT ZUR ERLANGUNG DES GRADES
DOKTOR-INGENIEUR (DR.-ING.)
VON DIPL.-INFORM. FLORIAN-WOLFGANG STOCK AUS
BRAUNSCHWEIG

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik

Referenten:	Prof. Dr.-Ing. Andreas Koch
	Prof. Dr.-Ing. Christian Hochberger
Einreichung:	2018-02-05
Prüfung:	2018-03-19

Darmstadt – Februar 2018
D 17

Stock, Florian-Wolfgang : *Rapid Prototyping and Exploration Environment for Generating C-to-Hardware-Compilers*
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
Tag der mündlichen Prüfung: 2018-03-19

ABSTRACT

There is today an ever-increasing demand for more computational power coupled with a desire to minimize energy requirements. Hardware accelerators currently appear to be the best solution to this problem. While general purpose computation with GPUs seem to be very successful in this area, they perform adequately only in those cases where the data access patterns and utilized algorithms fit the underlying architecture. ASICs on the other hand can yield even better results in terms of performance and energy consumption, but are very inflexible, as they are manufactured with an application specific circuitry. Field Programmable Gate Arrays (FPGAs) represent a combination of approaches: With their application specific hardware they provide high computational power while requiring, for many applications, less energy than a CPU or a GPU. On the other hand they are far more flexible than an ASIC due to their reconfigurability.

The only remaining problem is the programming of the FPGAs, as they are far more difficult to program compared to regular software. To allow common software developers, who have at best very limited knowledge in hardware design, to make use of these devices, tools were developed that take a regular high level language and generate hardware from it.

Among such tools, C-to-HDL compilers are a particularly widespread approach. These compilers attempt to translate common C code into a hardware description language from which a datapath is generated. Most of these compilers have many restrictions for the input and differ in their underlying generated micro architecture, their scheduling method, their applied optimizations, their execution model and even their target hardware. Thus, a comparison of a certain aspect alone, like their implemented scheduling method or their generated micro architecture, is almost impossible, as they differ in so many other aspects.

This work provides a survey of the existing C-to-HDL compilers and presents a new approach to evaluating and exploring different micro architectures for dynamic scheduling used by such compilers. From a mathematically formulated rule set the *Triad* compiler generates a backend for the *Scale* compiler framework, which then implements a hardware generation backend with described dynamic scheduling.

While more than a factor of four slower than hardware from highly optimized compilers, this environment allows easy comparison and exploration of different rule sets and the micro architecture for the dynamically scheduled datapaths generated from them. For demonstration purposes a rule set modeling the *CO-COMA* token flow model from the *COMRADE 2.0* compiler was implemented. Multiple variants of it were explored: Savings of up to 11 % of the required hardware resources were possible.

ZUSAMMENFASSUNG

Heutzutage gibt es eine immer größere Nachfrage nach mehr Rechenleistung, bei gleichzeitigem Wunsch immer weniger Energie dafür aufzuwenden. Momentan sind Hardwarebeschleuniger die beste Lösung hierfür. Während GPUs in diesem Gebiet sehr erfolgreich sind, bringen sie ihre beste Leistung nur zur Geltung, wenn die Algorithmen und Speicherzugriffsmuster auf die zugrundeliegende Architektur abgestimmt sind. Anderseits können ASICs noch mehr Leistung bei noch geringerem Energieverbrauch zur Verfügung stellen, sind aber aufgrund ihrer festgelegten Funktionalität sehr unflexibel. Eine Kombination aus beiden Ansätzen sind FPGAs: Sie können bei hoher Energieeffizienz eine große Rechenleistung zur Verfügung stellen, sind aber gleichzeitig durch ihre Rekonfigurierbarkeit flexibler als ASICs.

Ein offenes Problem ist aber immer noch die Programmierung der FPGAs, da sie viel schwerer zu programmieren sind als herkömmliche Software. Eine mögliche Lösung hierfür sind C-to-HDL Compiler, die herkömmlichen C Code in eine Hardwarebeschreibungssprache übersetzen, um daraus Hardware zu generieren. Viele von diesen Compilern haben Einschränkungen was den unterstützten Sprachumfang angeht, und unterscheiden sich in den verwendeten Optimierungen, der Ablaufplanung, der generierten Mikroarchitektur, ihrem Ausführungsmodell oder der Zielhardware. Diese vielen Unterschiede machen einen Vergleich bezüglich nur eines Aspektes fast unmöglich.

Diese Arbeit bietet eine in die Breite gehende Übersicht über die existierenden C-to-HDL Compiler und stellt ein System vor, das eine schnelle Evaluierung verschiedener Ansätze zur dynamischen Ablaufplanung ermöglicht. Hierzu liest der Compilergenerator *Triad* einen formalen Satz Regeln ein, aus denen dann ein Compilerbackend für das Compilerframework *Scale* generiert wird, das C in eine Hardwarebeschreibungsprache übersetzen kann. Die erzeugte Hardware nutzt dabei eine dynamische Ablaufplanung, die durch den formalen Regelsatz definiert wurde.

Während die generierte Hardware mehr als viermal langsamer ist, als die von spezialisierten optimierenden Compilern, erlaubt die vorgestellte Umgebung das schnellere Ausprobieren von verschiedensten Ansätze. Zu Demonstrationszwecken wurde im Regelsatz die Ablaufplanung vom *COMRADE 2.0* Compi-

ler nachgebildet. Mit nur wenig Aufwand wurde eine Variante erkundet, welche bei Tests bis zu 11% weniger Hardware Ressourcen benötigt.

DANKSAGUNG

Hello, Hello – this is dedicated to the ones I love.

Edgar — *Electric Dreams*, 1984

Ein erster Stelle möchte ich meinem Doktorvater Prof. Dr.-Ing Andreas Koch danken, der mich immer gut unterstützt hat. Sowohl bei wissenschaftlichen als auch bei nicht-wissenschaftlichen Themen war er immer eine Hilfe und hat mich angetrieben als es nötig war.

Außerdem möchte ich Prof. Dr.-Ing. Christian Hochberger für das kurzfristige Erstellen des Zweitgutachtens danken.

Tobias Riepe habe ich, neben Andreas, für die vielen sprachlichen Verbesserungsvorschläge zu danken.

Mein größter Dank gilt meiner Familie für die Unterstützung. Insbesondere meiner Frau, die mehr Geduld mit mir hatte, als jede(r) Andere und immer für mich da war.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS AND ABBREVIATIONS

ACS Adaptive Computing System

ALU Arithmetic-Logic-Units

ASAP As-soon-as-possible

ASIC Application Specific Integrated Circuit

ASIC Application Specific Integrated Circuits

AST Abstract Syntax Tree

AT *activate* tokens

BB Basic Block

Behaviour Description Language BDL

CDFG Control Data Flow Graph

CDFG Control Data Flow Graph

CFF Control Flow Frame

CFG Control Flow Graphs

CLB Configurable Logic Block

CMDFG Control Memory Data Flow Graph

COCOMA COMRADE Controller Micro-Architecture

COINS COmpiler INfraStructure

CPLD Complex Programmable Logic Devices

CPU Central Processing Units

CSP Communicating Sequential Processes

CT *cancel* tokens

CTL CHiMPS target language

DF Dominance Frontier

DFG Data Flow Graph

DFGs Data Flow Graphs

DSL Domain Specific Language

DSP Digital Signal Processor

EBNF Extended Backus-Naur Form

Event-Condition-Action ECA

FPGA Field Programmable Gate Array

FSB Front Side Bus

FSM Finite State Machine

GPGPU General Purpose Graphic Processing Unit

HDL Hardware Description Language

HLL High Level Language

HLS High Level Synthesis

HPC High Performance Computing

HW hardware

ILP Integer Linear Programming

IO Input-/Output

IR Intermediate Representation

KPN Kahn-Process-Networks

KPN Kahn-Process-Networks

LB Loop Body

LP Linear Program

LSQ Load Store Queue

LUT Look Up Table

MUX Multiplexer

PDF Post-Dominance Frontier

RC Reconfigurable Computing

RCU RC unit

RTL Register Transfer Language

RTL Register Transfer Logic

RTL Register-Transfer-Logic

Scale Scalable Compiler for Analytical Experiments

SIMD Single Instruction Multiple Data

SSA Static Single Assignment

STL Standard Template Library

t-structured top-structured

VLIW Very Large Instruction Word

1 INTRODUCTION

> [...] *it's my opinion that anyone who can possibly introduce science to the nonscientist should do so.*
>
> ISAAC ASIMOV – Interview, 1980

Ongoing advances in computer technology allow for smaller and smaller manufacturing process sizes.

More and more transistors per area are available (as forecast by Moore's Law [135]) but despite the fact that Central Processing Units (CPU) utilize them, the top clock frequencies of them have stagnated since the mid-2000s around the 4 GHz mark (see Figure 1.1).

While some approaches use a smaller transistor size to keep the number of transistors constant, and fit the circuit into a smaller

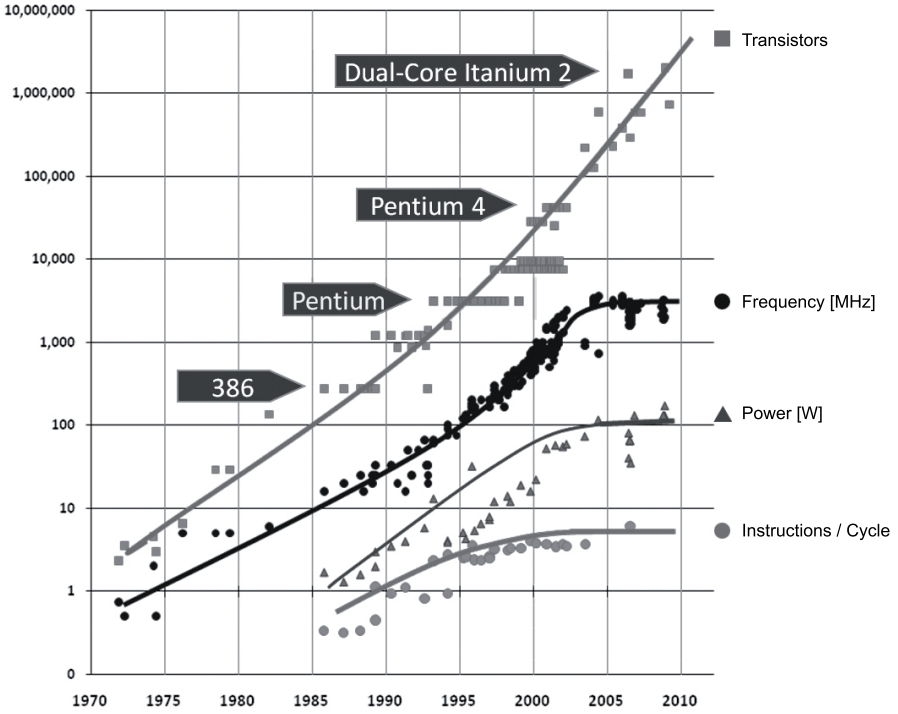


Figure 1.1: Development of clock frequencies and transistor count of conventional CPUs (data from K. Olukotun, Intel).

area, resulting a higher yield and lower power consumption, most approaches try to accelerate computations by utilizing the higher number of available transistors.

Recent approaches to utilize the higher number of the available logic resources concentrate mostly on

- bigger caches,
- multiple functional units, including multi- and many-core approaches as well as vector units, and
- dedicated task specific logic (for example encryption or video acceleration). Depending on the size, this could be just a small part of a chip, dedicated to one task, or even an own chip (Application Specific Integrated Circuit (ASIC)).

If more processing power is required, the first approach is no viable solution. Since computation time is an order of magnitude faster than memory access time, caches are used to hide the latencies by caching the data in faster, local memories. Thus bigger caches can improve memory bottlenecks in certain cases, but not in all of them. In particular, a single task does not benefit any more from further increased cache size after a certain, task-specific, limit is reached.

Replicating functional logic blocks works very well when the processor has to handle multiple tasks in parallel. Accelerating just a single task is only possible when the problem can be divided into multiple parts which can actually run independently from each other and thus utilize the replicated logic. This is entirely dependent on the input program, and in reality it ranges from problems that behave very nicely and allow for partitioning into very many parallel parts, to the other end of the spectrum with problems that are purely sequential and do not allow for parallel execution at all. A good example for the first group of problems are almost all math problems involving matrices and vectors, where operations are performed column-/element-wise independently from the other columns/elements. The modern cryptographic hash function `scrypt` [149] is an example of a function that was explicitly constructed to allow no parallelism in order to reduce the attack surface for brute force attacks.

In general Amdahl's Law [10] describes the upper limit of how beneficial parallelism can be (for example, a program which allows 50% of it to be executed on 2 units while the rest is limited to sequential execution, gains at most a speed up of 25% from parallelization).

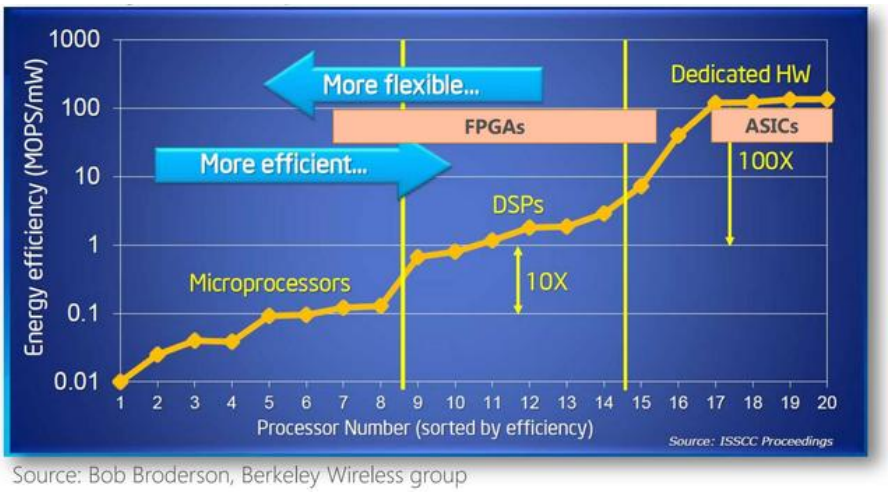


Figure 1.2: CPU, FPGA, and DSP comparison

Standard CPUs try to use the program's inherent parallelism via replication of small functional units (for example multiple load-store-units, vector instructions). Other problem-specific processors can maximize this by replicating complete smaller processors. For example, General Purpose Graphic Processing Unit (GPGPU) have hundreds of processors working on the same problem in parallel, usually in a Single Instruction Multiple Data (SIMD) design. Even though these units provide huge amounts of computational power (NVIDIA Tesla V100 has up to 30.0 TFLOPS [50]) this approach suffers from the lack of generality as not all problems can be mapped to the computation model of these GPGPUs.

Therefore, the third approach is the most viable when trying to accelerate a generic single task: Use hardware (HW) that is tailored to that specific task. The drawback with common dedicated task specific logic is, that it is dedicated. It is hardwired to its specific task. In case of changing tasks the hardware wastes just the area resources in best case, while in the worst case also power is wasted as well.

A solution to this is reconfigurable hardware. While not as fast as dedicated logic, it is usually faster than an all purpose CPU (but not as flexible, see Figure 1.2).

To get the best of both worlds; flexibility of a CPU, efficiency of an ASIC – one combines a regular CPU with reconfigurable hardware as accelerator. Such a combination is called an Adaptive Computing System (ACS).

1.1 ACS

In an ACS reconfigurable logic, usually in form of a Field Programmable Gate Array (FPGA), is combined with a conventional CPU.

FPGAs consist of very regular logic building blocks that are connected via routing resources. Building blocks that can implement different logic functions (usually via a Look Up Table (LUT)) and configurable switches connecting them allow the FPGA to realize almost arbitrarily complex logic functions. As mentioned above , ongoing miniaturization allows for more and more such blocks, and the trend is to use the high available number of transistors for more complex logic blocks (like complete Digital Signal Processing (DSP)-blocks). The commercial FPGA market is dominated by two big companies, Xilinx and Altera (has been owned by Intel since 2015).

Both offer FPGA series that contain a complete standard CPU as building block. Xilinx started in their old Virtex-4FX/-5FX series [196, 197] with an embedded PowerPC 405/440 hardcore and continued this trend with ARM dual-core Cortex-A9 in Zynq-7000 series and ARM quad-core Cortex-A53 in the newest Zynq UltraScale series[198].

Altera had an ARM v4 in their older Excalibur series of FPGAs [46], and have, similar to Xilinx, an ARM Cortex-A9 in their newest Arria and Cyclone [48, 49]) series.

On those models the CPU is on the same die as the FPGA, which allows a very fast on chip bus connection between both.

Such a close connection between FPGA and CPU is not necessary for an ACS; other architectures are possible, but the slower the communication between both is, the faster the FPGA must be to compensate for the data transfer overhead to generate a speedup.

At the moment solutions with connections on every level exist. From the FPGA CPU combination mentioned above, which use an on chip bus, to systems that use faster system buses between different dies within one package (for example Intel QuickAssist QPI FPGA Platform uses QPI links [40]) and system buses between different chips (e.g. Convey uses the Intel Front Side Bus (FSB) [42, 43]). The slowest variant are periphery buses (for example an FPGA card with PCI Express).

The connection between both is one factor that impacts the acceleration, while the other important factor is the memory model

(memory bandwidth, shared memory, cache coherence, virtualization, ...).

Such ACS already have shown their usefulness at a multitude of different applications, for example

- Cray built 2004 their entry level High Performance Computing (HPC) machine XD1 as ACS. An XD1 Rack consists of 12 chassis, each holding 12 64-bit AMD Opteron 200 CPUs accompanied with 6 Virtex II-Pro FPGAs [100].
- Convey/Micron built the HPC ACS systems $HC-1$ and $HC-1^{ex}$. A standard Intel Xeon CPU is coupled via FSB to 4 Xilinx Virtex 5 (in $HC-1$) / Virtex 6 (in $HC-1^{ex}$) FPGAs using a cache-coherent NUMA architecture. In the latest Convey generation the FPGA is coupled to the CPU via PCIe [42, 43].
- The HARP (and HARP 2) program from Intel, which introduces the *Intel QuickAssist QPI FPGA Platform*. On this platform an Intel XEON-CPU is coupled with an Altera FPGA via QPI and packaged together [40].

1.2 CHALLENGES

When using ACS new problems appear, namely:

- Communication between FPGA and CPU can be so expensive that all time gained by the acceleration from the FPGA is lost through Input-/Output (IO). Faster signaling can help here ([118]).
- The shared memory must be handled with care. Especially cache coherency must be considered.
- For optimal performance, the task has to be partitioned into a part that can be accelerated on the FPGA and a part that remains on the CPU. This partitioning problem, which has to split the program into those two parts, and modeling the communication from the chosen architecture as constraints, is from a theoretical viewpoint NP-hard. Solutions can use good heuristics (e.g. Kerninghan-Lin algorithm [128], genetic algorithms [59], simulated annealing or tabu search [61]), as well as exact solver (e.g. via Integer Linear Programming (ILP) [141, 142], branch-and-bound [21, 127] or dynamic programming [126]). Often this step

is avoided by requiring the programmer to manually decide which parts of the program have to be accelerated on the FPGA.

- The implementation of the reconfigurable logic in a Hardware Description Language (HDL).

The description of a synthesizable logic circuit in a HDL is quite different from programming in a conventional (procedural, functional or object orientated) language. It requires much more knowledge and experience and is very error prone.

To allow common software developers to develop for the ACS, the hardware description must be hidden from them. This problem is not specific to ACS; in pure HW development it is desirable to move away from low level HDLs to more abstract languages as well. Easing the programming and debugging for software developers and giving them the capability to develop HW is not the only benefit of this approach; a higher abstraction level also allows for faster turn around times for HW developers.

Besides extensions to standard HDLs to increase the level of abstraction (e.g. SystemVerilog), translating high level languages into HDL became a common approach (for more on these topics see Section 3.1).

To translate legacy code and make a high-level-to-HDL-compiler usable for as many software programmers as possible, it would be desirable to choose a widespread language as a source language. According to different statistical sources ([169, 190]) one of the most important and popular languages is still C ([111]). Only Java can compete with its popularity, but C is, due to its much simpler language constructs, much more suitable for translation into a HDL.

1.3 THESIS

In the Embedded Systems and Applications Group of the TU Darmstadt several different high-level-to-HDL-compilers were used and developed. Different research projects had different focuses, ranging from different input languages and different execution models of the generated hardware, via different compiler internals, to different memory architectures and much more.

Working with them and experimenting with different setups was (and still is) very tedious and trying to evaluate several micro architectures is often impossible, because the different projects often use different compilers and compiler frameworks.

Often it is only guesswork why two approaches yield different results, as one cannot trace back the different results to differences in used compilers, compiler settings, compiler optimizations, compiler internals, generated micro architecture, used hardware, environment or scheduling of the code.

The contribution of this thesis is to provide a system that allows the analysis and comparison of different micro architectures without the need to reimplement them completely anew in a common environment. To allow this, a framework is developed which is able to generate different C-to-HDL compilers, which have a common base but only differ in their HW generating backends, and hence allow an unbiased comparison.

The C-to-HDL compilers are generated from a formal description. This description is so versatile that it allows easy exploration of different architectures and concepts.

Not only can the compiler back ends be easily explored, but explorations and adaptions to the common compiler environments can be instantly applied to all existing generated variants by regeneration of the compiler, as well. This allows, for example, research of the impact of new compiler optimizations on different HW back ends.

The thesis is structured as follows:

First, the definitions and fundamentals used are introduced in Chapter 2.

Secondly, Chapter 3 will give a broad overview over prior and related work. An (non-exhaustive) survey of C-to-HDL-Compiler is done and they are classified according to a proposed taxonomy.

Although, the developed Compiler framework is independent from the C-to-HDL-Compiler COMRADE, it was one major motivation and it is analyzed in more detail.

In Chapter 4 the Domain Specific Language (DSL) used for the formal description is introduced and its implementation is described.

Afterwards the generated compilers are evaluated, together with the compiler generator itself, in Chapter 5.

Finally, the work is concluded with a summary and an outlook in Chapter 6.

2

FOUNDATIONS

Everything of any importance is founded on mathematics.

Jonny Rico – *Starship Troopers*, ROBERT A. HEINLEIN, 1959

As laid out in the introduction, the focus of this work is about the development of C-to-HDL-compilers. One of the goals of those compilers was to abstract the hardware, so software developers can use them. In consequence this means, that building such compilers requires knowledge from both worlds, hardware and software. Hence, this foundations section is divided into three parts. First, the software part: this is basically about compilers. The second part is about hardware generation and finally: co-execution of hardware and software, which is necessary when hardware should run as accelerator. In this chapter the used terms, formalism, and fundamentals for these three parts are defined. Most of the information in this section is just common text book knowledge, so an exhaustive list of references for them is omitted and only references to some of them are given ([1, 56, 110, 136]). The explanation of the terms is kept concise; the formal definitions are collected in Appendix A.

2.1 COMPILER

The typical compiler is split into three parts: front end, middle end, and back end. Usually the front end scans and parses the input, the middle end performs optimizations and the back end generates the output. The Intermediate Representation (IR) is

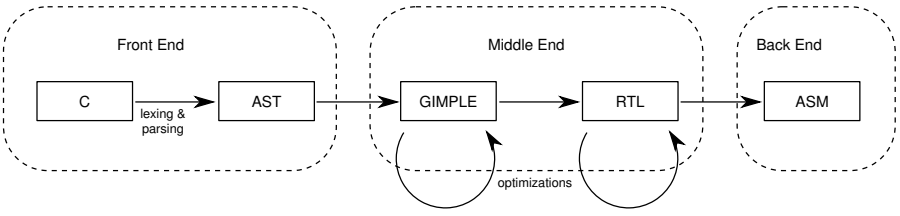


Figure 2.1: Compiler phases from the gcc Compiler

the internal data representation the compiler uses between different steps. Figure 2.1 shows a compiler having all three phases and using different IRs.

Several IRs are possible, some compilers even have multiple IRs, depending on the required level of abstraction (for example see [41, 121, 170]).

An often used IR is an abstract, assembler-like language. Depending on the intended optimizations and tasks, the abstraction level can vary. An example for a very abstract IR is LLVMs IR, which has, similar to a Turing machine, infinitely many registers. An example for a less abstract IR is the GCC Register Transfer Language (RTL) which is very close to the target machines assembler code. Figure 2.2 shows some examples of such IRs.

An often used data structure for IRs is the graph. These are commonly (additionally) used, especially for hardware compilers, to represent control flow Control Flow Graphs (CFG) and/or the flow of data in the program Data Flow Graphs (DFGs). The graph representation allows an easy combination of both pieces of information or the addition of other information in form of additional edges or nodes or annotations to them. A common enhancement are additional edges for memory dependencies. In the next sections these common graph IRs are explained.

2.1.1 *Control Flow*

The general idea behind a CFG, is that the program is modeled as graph, where the nodes correspond to the statements and operations in the program and the edges model the sequence of the possible execution order. The detailed formal definition is in Appendix A.

The non-branching blocks of code that become nodes are called Basic Blocks (BB) (see Definition 1).

Now we need to model the possible flow of execution, especially the possible branches. This is done with Control Flow Frames (CFF) (see Definition 2), which allows us to model the control flow in programs with a dedicated start and end node. A function *cond* is necessary to annotate edges of branches with the result of the condition. A special value ϵ is used to annotate default cases in switch statements and as value for non-branching edges.

```
int fak(int c) {
 if (c == 1)
   return 1;
 else
   return c * fak(c-1);
}
```

(a) Original C-Code

```
fak (int c)
{
  int D.1798;

  if (c == 1) goto <D.1796>;
  else goto <D.1797>;
  <D.1796>:
  D.1798 = 1;
  return D.1798;
  <D.1797>:
  _1 = c + -1;
  _2 = fak (_1);
  D.1798 = c * _2;
  return D.1798;
}
```

(b) High level GIMPLE IR from GCC

```
(insn/f 33 32 34 2 (set (reg/f:DI 6 bp)
        (reg/f:DI 7 sp)) "fak.c":1 81 {*movdi_internal}
     (nil))
(insn/f 34 33 35 2 (parallel [
            (set (reg/f:DI 7 sp)
                (plus:DI (reg/f:DI 7 sp)
                    (const_int -16 [0xfffffffffffffff0])))
            (clobber (reg:CC 17 flags))
            (clobber (mem:BLK (scratch) [0  A8]))
        ]) "fak.c":1 984 {pro_epilogue_adjust_stack_di_add}
```

(c) Low level GCC RTL (excerpt)

```
; Function Attrs: noinline nounwind uwtable
define i32 @fak(i32) #0 {
[...]
; <label>:7:
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @fak(i32 %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
[...]
```

(d) LLVM IR

Figure 2.2: Example of different IRs for the standard *factorial* example code.

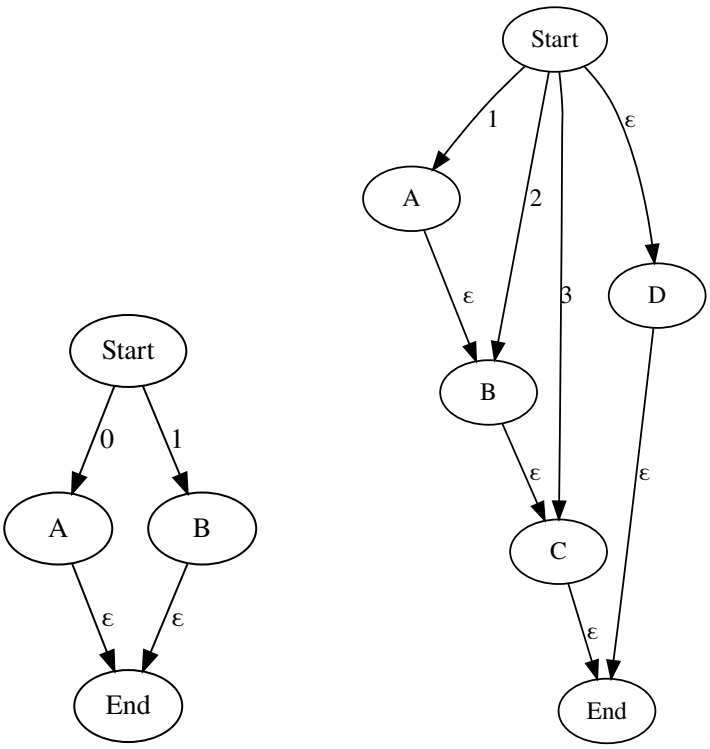


Figure 2.3: Examples of control flow frames (if and switch)

Figure 2.3 shows two graphical examples of two CFFs representing a single `for` and a single `switch` statement. In general they can be almost arbitrarily complex.

With the CFF supplying the structure of the graph, the addition of BB as nodes gives a complete graph, which models the complete program. This resulting graph is called *CFG* (see Definition 3).

For compiler optimizations and transformations that operate on the CFG, it is useful to define some terms which occur again and again to ease the description of the algorithms:

A *branch node* (see Definition 4) is a node where the control flow splits, and at a *join node* (see Definition 5) the previously split flow joins again. A subset of nodes is called a *region* (see Definition 6). A *dominator* (see Definitions 7 and 9) of a node is every node that must lie along the path from the start node to itself. Similar a *post-dominator* of a node (see Definitions 8 and 10) is every node that must lay along the path from the node itself to the end node. A node n_1 *controls* (see Definition 11) another node n_2, if n_2 is executed after n_1 and n_1 decides whether n_2 is executed or not.

```c
int main() {
  int i;
  printf("The loop starts...\n");

  for (i = 0; i < 5; i++)
     printf("Hello World!\n");

  printf("The loop has ended!\n");

  return 0;
}
```

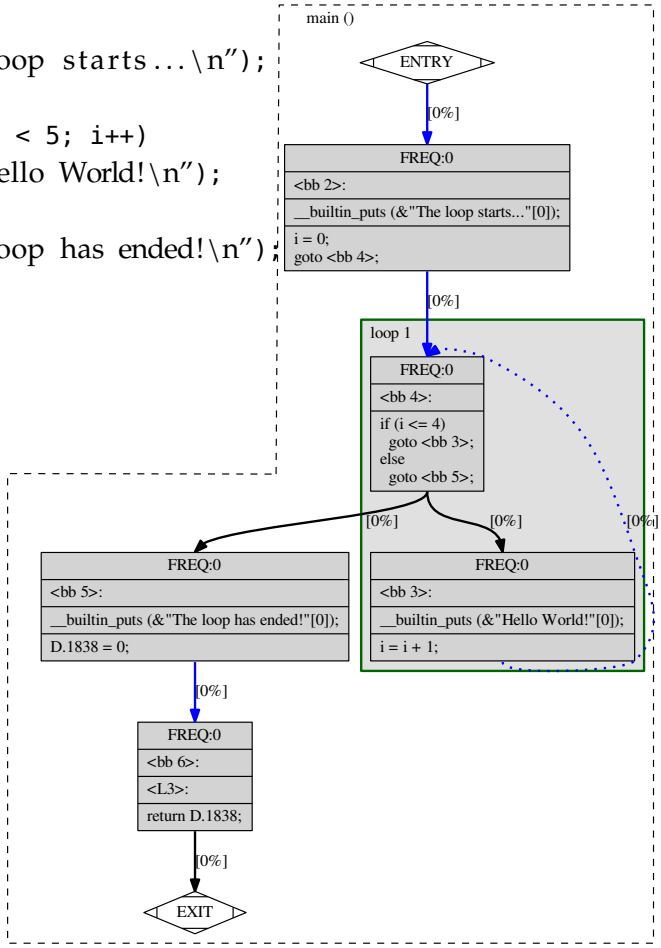


Figure 2.4: C-Program and the generated corresponding CFG.

Figure 2.4 shows a simple C program together with the CFG that represents this program. For graphical representation different degrees of detail are possible, such as giving the detailed statements of the BB in the nodes when required, or just labeling the nodes with a unique identifier when just the structure is relevant. In the given example each gray block represents a basic block, which is identified by a sequential number within the <bb > tag.

The above terms can all be found within the small example:

BRANCH NODE: The only basic block that is a branch node is <bb 4>. It is after the check of the loop end condition, where the control flow can either branch to the next loop iteration or to the next instruction after the loop.

(a)

```
int func(int a) {
  int s;
  s = 0;
  l1: s += a;
      a--;
  if (a != 0)
     goto l1:
  return s;
}
```

(b)

```
int func(int a) {
  int s;
  s = 0;
  do {
    s += a;
    a--;
  } while (a != 0)
      ;

  return s;
}
```

(c)

```
int func(int a) {
  int s;
  s = 0;
  s += a;
  a--;
  while (a != 0) {
    s += a;
    a--;
  }
  return s;
}
```

Figure 2.5: C program with gotos (a), transformed into a structured C program (b), and into a t-structured C program (c).

JOIN NODE: <bb 4> is also the join node. Before the check of the loop condition the control flows of the pre-loop code and in case of repeated loop iteration join.

REGION: As a region is just a subset, multiple regions are present, for example <bb 4> and <bb 5> make up a region (in Figure 2.4 boxed and labeled with loop 1).

DOMINATES RELATION: An example for domination would be <bb 2>, as it dominates every other node (i.e. it must be executed before).

POST-DOMINATES RELATION: Similarly <bb 6> post-dominates every other node.

(POST-)DOMINANCE FRONTIER: For node <bb 3> the post dominance frontier would contain only the node <bb 4>. In this case it would be also its dominance frontier.

CONTROL DEPENDENCE: <bb 4> controls <bb 3>.

When handling different input languages, different patterns are generated in the resulting CFG. If the input is restricted to structured imperative programming languages the resulting graphs have some advantageous properties. Structured programs are all imperative programs that have no explicit or implicit goto (break, continue, exception or similar) statements and are just a concatenation, selection or repetition of instructions

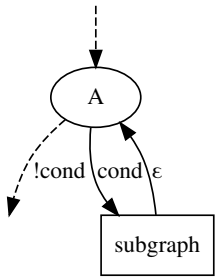


Figure 2.6: Subgraph containing a loop that is t-structured.

or subprograms. All common C programs can be transformed via goto removal into such a structured program [23, 64]. As a further simplification the structured program can be normalized into a top-structured (t-structured) program (see Definition 12). This consists only of loops that are evaluating the loop condition at the top, and not at the bottom.

The C programming language has just a `do-while`-loop with evaluation at the bottom, which can be easily transformed into a functional equivalent `while`-loop (see Figure 2.5 for such transformations).

Such t-structured programs have the useful property that each node of the CFG has at most one controlling node [67]. When generating a hardware description for the program this simplifies the task, as the translation/generation is limited to one kind of loop.

For later discussion we also introduce the following terms: *Back edge* (see Definition 13), loop header (see Definition 14), and loop body (see Definition 15).

Another advantageous property often used is the Static Single Assignment (SSA) form [53] (see Definition 16), which demands that each variable is defined only in one place. This property is not limited to CFGs, but can also be applied to all other IRs that contain assignments. It not only simplifies and improves many optimizations but is also advantageous for the hardware generation, as every data has exactly one single point of definition.

It is easy to transform arbitrary code into code that fulfills the first condition of the definition: Every time a variable is written after its first definition in the code, the variable is renamed into a new one. Of course all subsequent accesses have to be corrected to use the new name. A common technique is to use the same

```
x = x + y;
y = x + y;
z = x + y;
```

```
x1 = x0 + y0;
y1 = x1 + y0;
z0 = x1 + y1;
```

Figure 2.7: Transformation from code into SSA form.

```
x = ...
if (cond)
   x = x + y;
z = x + y;
```

```
x0 = ...
if (cond)
   x1 = x0 + y0;
z0 = x? + y1;
```

```
x0 = ...
if (cond)
    x1 = x0 + y0;
x2 = φ (x0, x1);
z0 = x2 + y1;
```

Figure 2.8: Regular C-Code, its SSA form without ϕ function (depending on the condition the $x_?$ must be either x_1 or x_2), and its SSA form with ϕ function.

variable name with an index (version number) for each definition. Figure 2.7 shows such a renaming.

The second property of the definition demands that after joining different code paths, both of which redefine the variable, a new, unique variable is created which can be used to access the current value afterwards, regardless of which branch was taken. Therefore, a function is introduced that selects one of its arguments, depending on the control flow that led to its execution. This function is called ϕ-function. In Figure 2.8 an example for the use of this ϕ-function is shown.

As no CPU physically supports such a function, all compilers which emit assembler code must remove these ϕ-functions. This process is often called *conversion out-of SSA form* or *destruction of SSA form*, even when it is afterwards still has the SSA property.

During the conversion the versioned variables are kept, and appropriate copy instructions are inserted in the different control flows before the ϕ-function. Figure 2.9 shows the example SSA form from Figure 2.8 after such a destruction. This example is trivial but in general, with more complex control flows, it can get quite complicated. An often used approach is the algorithm from Briggs et al. [25], which improves the algorithm originally presented [53].

2.1.1.1 *Data Flow*

Data flow models go back to the 1960s. They were introduced to model the flow of data items through a network. Various similar

```
x0 = ...
if (cond)
    x1 = x0 + y0;
    x2 = x1
else
    x2 = x0
z0 = x2 + y1;
```

Figure 2.9: Example code from Figure 2.8 converted out-of SSA form (unoptimized)

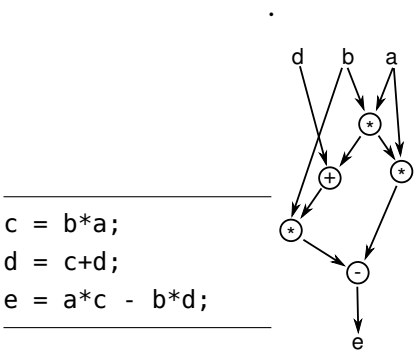


Figure 2.10: Example for the DFG representation of C-Code.

models, some super-sets of others, were proposed. Some are still in widespread use, like Petri Nets ([138]), computation graphs ([106]), or Kahn Process Networks ([105]).

Similar to the CFG, the data flow is also modeled as a graph, but in this case it is a directed acyclic graph. The nodes represent operators, while the edges model the flow of the data items between the operators.

Figure 2.10 gives an example of a DFG and the C code it represents. The elegant modeling of the creation of data, its consumption and the flow between operations is not only used inside compilers. It can also be used to represent Register-Transfer-Logic (RTL), which is used to describe synchronous hardware circuits. Even complete architectures have been designed on the data flow model ([82]).

In this simple form control flow cannot be handled, only a single basic block can be represented by a DFG. One way to include control flow would be to combine CFG and DFG representations into a Control Data Flow Graph (CDFG), where special edges are inserted into the DFG to represent the control dependencies (Figure 2.11 shows such a graph).

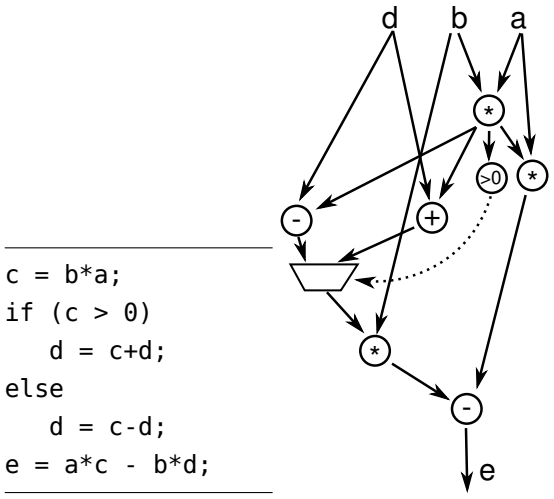


Figure 2.11: Example for the CDFG representation of C-Code.

```
a = b + c;
d = a + c;
```

(a) Read after write (RAW) dependency

```
a = b + c;
b = c + d;
```

(b) Write after read (WAR) dependency

```
c = a + b;
c = d + e;
```

(c) Write after write (WAW) dependency

Figure 2.12: Examples for memory dependency (assuming that each variable access is actually a memory access).

2.1.1.2 *Memory Dependence*

The focus of the COMRADE compiler[67] (more details in Section 3.3) was the Control Memory Data Flow Graph (CMDFG). These further advancement to CDFGs adds another type of edge, that is added between two nodes, describing memory accesses (read or write) when there is a dependency between those.

Memory accesses that are independent from each other have the advantage, that they can be executed in parallel or out-of-order.

Figure 2.12 shows the three kind of dependencies that exist. When in one of the cases (in listing (a) variable `a`, in (b) `b`, and in (c) `c`) the execution order is changed, a wrong value is read or written, invalidating the computation.

These cases are very easily to detect, far more problematic are the cases, when the access happen not via a constant reference, but with an arbitrary pointer, where the address is not known at compile time. When no other information is available, the compiler must assume the worst case and treat them as de-

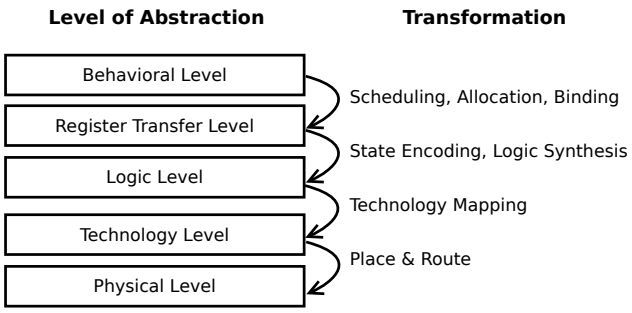


Figure 2.13: Flow for generating hardware.

pendent, presumably degrading the performance. Besides using hints from the programmer (the C language introduced for this reason the `restrict` keyword in the *C99* standard) many compilers try to improve the situation by trying to proof that two pointers do not refer to the same position (see Section 2.2.3.4).

In the *COCOMA* from *COMRADEs* CMDFG[67] the memory edges are not only used to model the memory dependencies between the memory accesses, but also to serialize them. Having only one memory backend port, the accesses to them must be ordered somehow. Using memory edges between a memory access and the next one in program order fulfills this task.

2.2 HIGH LEVEL SYNTHESIS

Hardware generation in general is a long and complex process. Figure 2.13 shows the most important steps involved when building digital circuits.

The first step, the transformation from the behavioral level to a structural description on the RTL, is called High Level Synthesis (HLS) [133]. Starting with a behavioral description (usually in an HDL like Verilog, VHDL, Chisel or Bluespec) of what the hardware is supposed to do, the level of abstraction is lowered with different transformations. Each of the transformations leads to a lower level of abstraction, with the transformations being: Allocation, binding and scheduling. As these are part of almost all HLS compilers, they are explained in greater detail in the sections below. The other steps happen after the HLS by specialized programs/tools, so just a concise explanation is given; for a more detailed description see [133, 153].

STATE ENCODING Generates a Finite State Machine (FSM) and the encoding for the required states when necessary.

LOGIC SYNTHESIS The RTL description gets translated into an logic-level representation (which is usually a netlist). Also includes a minimization/optimization of the generated logic.

TECHNOLOGY MAPPING Process of mapping generic logic functions to the specific functions available in the technology used.

PLACE & ROUTE In this step the available objects in the technology get physically placed and connected. Both are very complex, and each of it alone is NP-complete ([73, 74]).

Even when listed separatly, several of them are highly dependent on each other; especially, those that are shown on the same level in Figure 2.13 are often executed together. This applies even more to the transformations of the HLS step. These phases are highly intermingled: Each decision made in one phase has an impact on subsequent phases. For example if two operations are scheduled on the same time step, they cannot be bound to the same resource. Decisions which seem optimal for a phase can degrade the result quality of a later phase. As there is (so far) no per se perfect ordering of the phases, the problem is known as *phase order problem* or *phase coupling problem*. Most hardware synthesis systems apply the scheduling first, then allocation and binding, but other approaches have also been tried [166]. Some perform binding and allocation before scheduling, others try to combine scheduling, allocation and binding into a single phase (e.g. with ILP formulations) [133].

2.2.1 *Scheduling*

The starting time of each operation is determined in such a way that all constraints are met. There are as simple algorithms as As-soon-as-possible (ASAP) scheduling, which schedules an operation as soon as the constrains allow it. But there are also more sophisticated schemes, like the often used *List Scheduler*. For HLL compilers the used scheduling algorithms can be classified in one of three groups, depending on when and how the scheduling happens:

STATIC The exact fixed starting times are determined completely by the scheduling algorithm (like the above mentioned ASAP or List scheduler) at compile-/design-time.

DYNAMIC The necessary timings are not computed statically at compile-/design-time, but the necessary decisions when an operation has to be started are made at run-time.

MIXED Sometimes also referred as quasi-static, this is a combination of static and dynamic scheduling, where some of the scheduling decisions are made at run-time and some at compile-/design-time.

For the small example problem the operations are just sequentially executed. But with more complex computations the scheduling of the operations is far more critical as this can impact the latency of the whole computation.

2.2.2 *Allocation and Binding*

In the allocation step it is defined which resources will be used in the final circuit. During the binding step the mapping from the operations to the allocated resources happens.

As both steps depend on each other (only allocated resources can be bound on one hand, on the other it makes only sense to allocate resources that become bound later), they are performed together in one phase.

This is more complex than it seems, as there is not necessarily a one-to-one mapping from the operations to the hardware resources. A resource could be shared between different usages, it could even perform different operations (for example an ALU can do different computations). For operations the concrete implementation must be chosen (many operations allow different implementations which trade area for speed).

In general this is an optimization problem which usually aims at minimizing the used resources and/or the required glue logic and wiring, or more concretely the used area. Like many other problems in the hardware generation flow this problem is NP-complete, so different heuristics are used to solve it[166].

Example

As an example for the three HLS tasks a complex multiplication $(a + bi) * (c + di) = e + fi$ is used. Figure 2.14 shows the DFG for this computation. To make the allocation not too trivial, limited resources are assumed: only two multipliers, one adder, and one adder/subtractor are available and are assigned as in Figure 2.15.

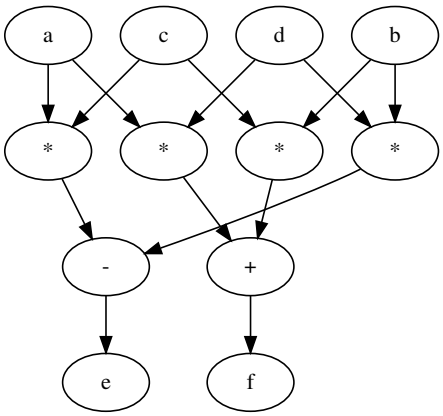


Figure 2.14: DFG for the complex multiplication $(a + bi) * (c + di) = e + fi$.

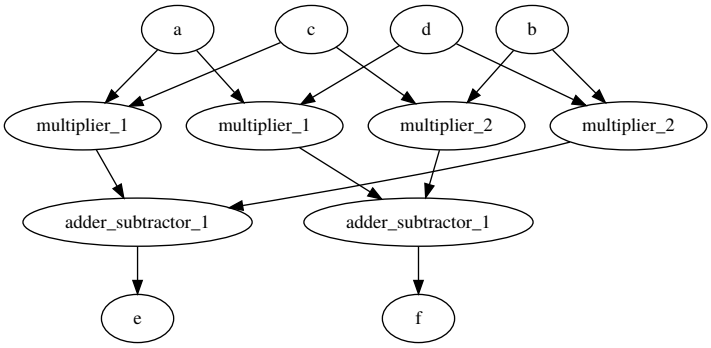


Figure 2.15: Binding of 2 multiplier, 1 adder, and 1 subtractor to the DFG from Figure 2.14.

After the allocation and binding an ASAP scheduling is performed. The algorithm is quite simple: Nodes are inserted in topological order of the dependency graph into the first possible time slot, so that a) the previous nodes are finished, and b) the required resources are available.

In the example the order of the nodes is
`multiplier_1` (with inputs from a and c),
`multiplier_1`(a, d),
`multiplier_2`(c, b),
`multiplier_2`(d, b),
`adder_subtractor_1`$(-)$,
`adder_subtractor_1`$(+)$.

The first multiplication by `multiplier_1` (with inputs from a and d) can be scheduled on time slot 1 without problems. While a second multiplication can be allocated on the other multiplier, the third multiplication is allocated to `multiplier_1` too. So this multiplication is scheduled on time slot 2 (for the sake of sim-

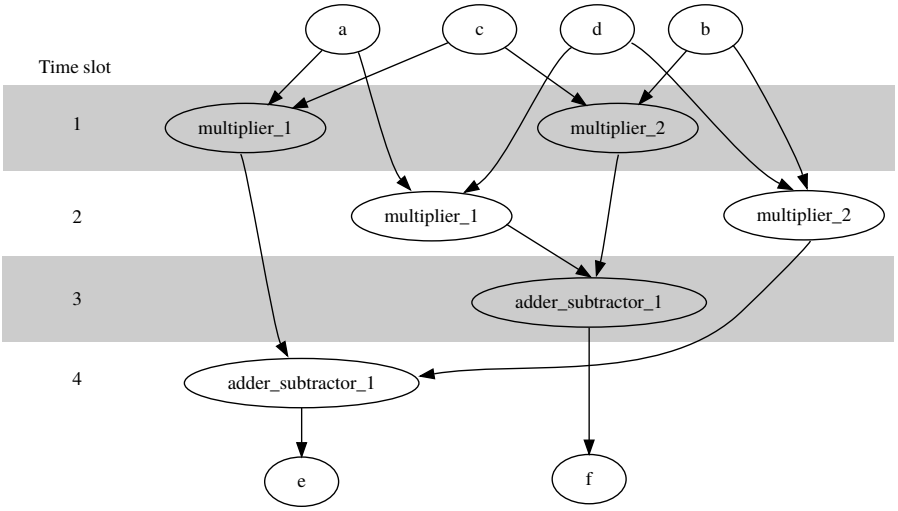


Figure 2.16: Schedule from the operations from Figure 2.15.

plicity of the example it is just assumed that all operations take one time step).

In the same way `multiplier_2` got placed on time slots 1 and 2. The `adder_subtractor_1` depends on the last scheduled multipliers and so the first operation is scheduled on time slot 3, and the last afterwards on time slot 4. The resulting schedule is depicted in Figure 2.16.

The ASAP schedule is not the perfect solution even in this small example. Figure 2.17 switches the insertion of the multiplier operations on `multiplier_1` and as a consequence the computation finishes one time slot earlier.

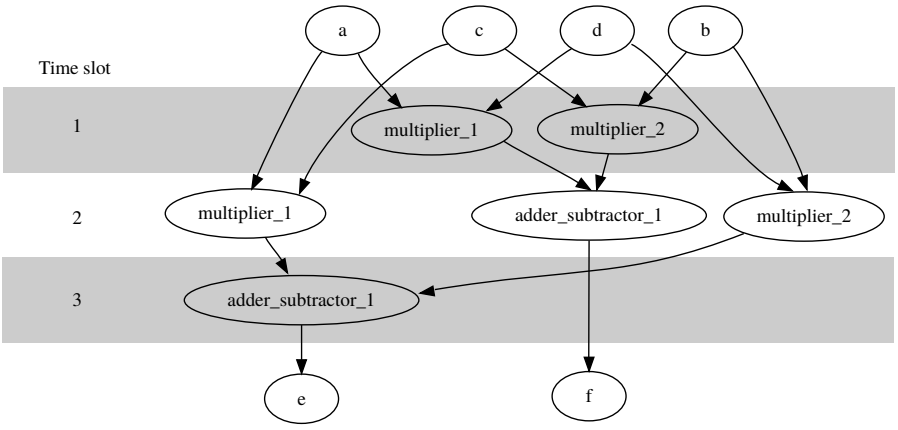


Figure 2.17: Improved schedule of Figure 2.16.

2.2.3 *Generating Hardware from High Level Languages*

Automated hardware generation from a higher level of abstraction, such as a regular programming language, is possible in a number of ways. The most common way is to perform the HLS steps and lower the abstract high level language to an HDL. This way, everything which is close to hardware and dependent on the actual technology used (e.g. logic synthesis, mapping, place and route) is dealt with by the regular tool flow tools.

Despite the most common approach, it is not the only one. Two other ways are:

1. Use libraries in the HLL and use those to perform steps in the hardware generation process below the HLS. One example for this is the *JBits* library from Xilinx for Virtex 2 Pro [193–195]). It enables the programmer to write the configuration bitstream for a reconfigurable device directly and gives him direct control over the used hardware resources and even placing and routing.

2. Generate microcode for a generated/selected application specific or configurable processor. The hardware generation is reduced to selection, configuration, and programming of a processor (an example for this kind would be *PICO* [108]).

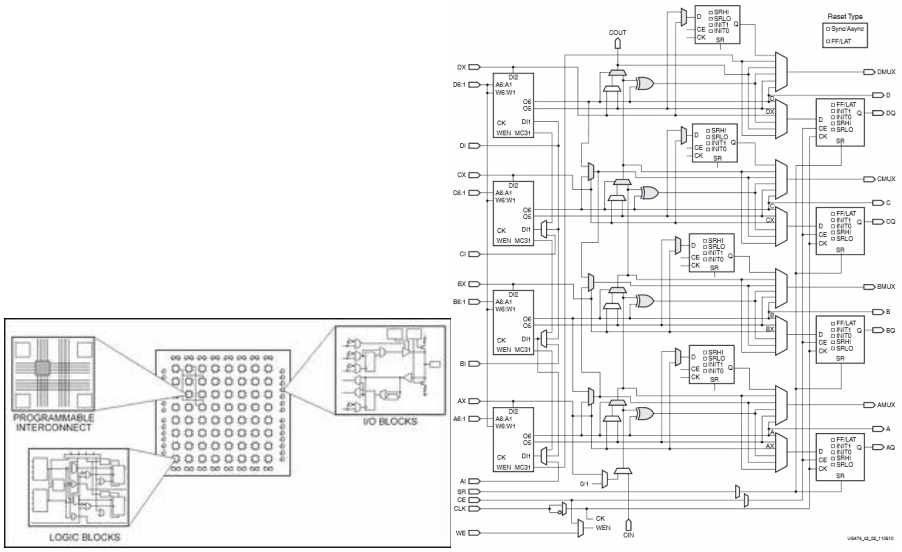
Chapter 3 and Appendix B list different compilers and the methods used for hardware generation more detailed.

This work takes the same approach as most of the other compilers and the compiler generator creates compilers that generate hardware by emitting Verilog HDL code.

2.2.3.1 *Target Hardware*

While HLS can target any kind of hardware, the most common application is to generate hardware descriptions for *reconfigurable hardware* (in opposite to Application Specific Integrated Circuits (ASIC)).

The main element of Reconfigurable Computing (RC) is a hardware element whose functions can be reconfigured, a configurable device. Most widespread devices that allow such a reconfiguration are Complex Programmable Logic Devices (CPLD)s and the similarly constructed, but bigger and more complex FPGAs.



(a) Schematic FPGA

(b) A *slice* (a CLB has 2 of these) of a Xilinx Virtex 7

Figure 2.18: Xilinx FPGA (source Xilinx).

These consist of different configurable elements which can be connected in different ways. Configurable elements not only include computational or logic elements but also I/O elements and memory.

Depending on the granularity of the device, the size of the computational reconfigurable element changes. Coarse grained devices can have complete Arithmetic-Logic-Units (ALU)s as computing elements (an example for a coarse grained architecture is PACT XPP[15]). On the other hand fine grained devices contain LUTs and/or multiplexers as the smallest logic element. The LUT can be configured as arbitrary n-input binary function. Depending on the vendor and the technology used n is usually 4 or 6 (the two vendors with the greatest market share Altera and Xilinx use this approach). Figure 2.18 shows such a modern, newest generation FPGA from Xilinx, a Xilinx Ultrascale. The biggest models have up several hundreds of thousands of Configurable Logic Block (CLB)s, each containing 8 LUTs and 16 flip-flops. Furthermore, they have tens of Megabits of configurable RAM blocks, thousands of DSP hard blocks and several hard blocks for I/O (for example Ethernet, PCIe, Interlaken)[1]. Some models even include multiple ARM CPU hard blocks. Arranged in a regular grid between these configurable elements are channels that contain the routing resources. The wires can

1 The exact numbers depend on the model and can be found in [200]

be configured to connect to the logic and as to how they connect with each other to route the nets. For clock signals dedicated networks are available as well as clock management resources.

The reconfigurability of the device comes at a price: While ASICs can clock up to several GHz, the newest generation FPGAs are much slower. The above mentioned UltraScale FPGA has as a maximum frequency f_{max} for its element of 525 - 741 MHz (depending on the type of element and speed grade of the device) [199]. While there are devices that have persistent configuration memory (for example ACTELs Flash-based FPGAs), most of the bigger devices are SRAM based. This means they are no instant-on devices, as the configuration must be read before operation, and they struggle with the same problems as SRAM; esp., they are sensitive to ionizing radiation [173].

While not reaching the frequencies of an ASIC, the FPGA can still accelerate many computations compared to a CPU due to application specific hardware acceleration. Additionally, the reconfiguration allows for a change of the implemented hardware afterwards (for example to changing standards or protocols, or when an error is found), and for short development cycles while developing hardware. When the total number of produced units is small, the FPGAs are also much cheaper than an ASIC.

While RC can accelerate computations, an RC unit (RCU) is very inefficient when handling general-purpose applications. The typical application spends most of their time in a small computational part, while the rest are I/O and management tasks, which are handled very well by modern CPUs.

The solution is the hybrid approach already mentioned in Chapter 1, called ACS, that consists of a standard CPU, for the administrative tasks, accompanied by an RCU, that handles the compute intense parts. This combination is so popular that the FPGAs do not need to be paired up with an external CPU[2], that the manufacturers offer FPGAs with CPU cores (and CPUs with FPGA-fabric).

The main drawback for the ACS is the knowledge required on the programmer's side. Despite using an HDL, the development of hardware is different from plain software programming. At this point the HLS-compiler is supposed to bridge the knowledge gap for the software developer so he can utilize the RCU without expert HW design knowledge. At least that is the hope; in reality there are many problems.

2 or the CPU with an external FPGA, depending on the fact if you see the whole system FPGA-centric or CPU-centric

```
if (a+c > b)
   c = a*b;
else
   d = c/d;
```

Figure 2.19: The hardware version of this high level code requires +, * and / operator, but never * and / simultaneously.

2.2.3.2 *Difficulties*

So far, the flow for generating hardware from a high level language presented seems fairly easy: translate it into an HDL, and let the HDL compiler (and other tools) handle the complicated tasks from there. From a theoretical point-of-view the problem that arises is, that the RTL synthesizeable subset from the HDL is not Turing-complete. It can be used for describing netlists, like in XML for describing Turing-complete hardware, but it is itself not Turing-complete. Translating one structured high level language into another is usually done by replacing each language construct (sequential execution, loop, conditional execution, ...) with the corresponding construct of the target language. HDLs are not structured, and due to this mismatch, the translation of the high level language into the low level HDL is not easy. For instance, neither the subprogram mechanism (in a modern sense, where arbitrary calls are allowed) nor does the loop construct available in almost every high level language does exist in HDLs. Instead of writing a loop itself, a circuit which will later implement the loop and dedicated control logic has to be created. This process is a complex process, even for one loop, but with nested loops this gets increasingly more complex.

Several compilers avoid this problem by just translating the innermost loop. The computation inside this innermost loop is translated into a data path, and the generated hardware is used as a streaming kernel[3] (for example [120] uses this approach).

Another drawback is that the hardware which should be equivalent to a software program has to supply all possibly necessary functions, even when not all are neededed at all times (or not need at all). It is just not possible to remove one subroutine

3 Streaming is a parallel computation paradigm, where data is considered to be a stream of data and the computation is performed on each of the data elements. No further allocation or communication besides the *stream* is required, and usually pipelined hardware profit from this approach, as the pipeline is kept filled by the stream.

(or parts thereof) in hardware and replace it with a different one. Self reconfiguration would allow this, but no HDL assumes or supports it.

Thus, in many cases complete programs cannot be implemented in hardware. One solution is to keep the program as a software program on a CPU, and just accelerate speed critical routines in hardware (these routines are often called kernels). The controlling software on the CPU manages in- and output and configures that kernel onto the FPGA that is required for the current program (ACS).

This execution in hardware and software together is called Hardware-Software-Co-Execution (HW-SW-Coexecution). It leads to another NP-complete problem: the partitioning.

2.2.3.3 *Hard- and Software Co-Execution*

In an environment where hardware and software are executed together, the program must be *partitioned*. This is the process of selecting which parts of the software are executed in hardware and which stay on the CPU.

The most important task is to identify the part of the program which should be moved to the accelerator. Usually, only compute intense parts are handled by the accelerator, parts which require limited computation or are I/O-intensive are kept on the CPU. The reason for this is that the specific hardware is often not capable of performing those operations efficiently or cannot do it at all (e.g. system-calls or other I/O-tasks).

Due to the NP-completeness of the problem, many different approaches have been implemented, which are also heavily influenced by the target architecture and computation model.

The different approaches can vary in several aspects, namely:

GRANULARITY Depending on the architecture and the available size for the generated hardware the granularity of the accelerated part can vary from a single instruction to complete programs.

MANUAL VS AUTOMATIC The method for identification of the parts which may and/or should be run in hardware. Many systems rely on manual annotation in the high level code, few try to automatically decide which parts become hardware. Most automatic approaches use some kind of profiling, where test runs are made in software only. While performing these runs, execution information (such as branch

probabilities, memory access patterns, problem sizes, execution times) is collected. Using a mathematical model, which takes configuration times, data transfer times between RCU and CPU, estimated and available hardware areas, and the collected data into account, the relevant parts are identified.

CHOOSE BETWEEN HW AND SW EXECUTION The decision whether the generated hardware accelerator should be used can be dynamic or static. In the static case it is always used (not using it is also an option, but then why it was generated in the first place?), but in the dynamic case the decision, is made at runtime. The software still contains the necessary software instructions for the generated hardware, and depending on several parameters, it is possible that the software version is executed, and the hardware is not used. It is also possible that both versions are run initially, and in later executions only the faster one gets selected.

CONTROL FLOW TRANSITION HW-SW Several systems allow only the CPU to start a computation on the RCU and expect the result later. This approach excludes all software that is not computation-only from acceleration. Even the simplest I/O, even when it is not supposed to happen regularly, prevents the routine from getting implemented in hardware, as the hardware cannot execute it. Figure 2.20 gives an example for such a scenario. More sophisticated systems allow the hardware to pause and perform software callbacks to execute the system- or I/O-functions ([107]).

AREA A hard limitation for the generated hardware is set by the available resources on the target reconfigurable device. While most approaches try to gain as much benefit as possible by using as much hardware as possible, some go even a step further. They do not only partition into CPU and RCU parts, but also perform temporal partitioning ([188]). This works best with algorithms that have different phases, which are executed sequentially.

The list of HDL compilers in Appendix B contains more details and references on the partitioning methods used by them.

```
double inverseSum(int count, int* numbers) {
  double result = 0.0;

  for (int i = 0; i < count; i++) {
      if (numbers[i] == 0) {
         printf("All numbers must be != 0!\n");
         return -1.0;
      }
      result += 1.0/numbers[i];
  }

  return result;
}
```

Figure 2.20: The `printf`, only intended for error handling, stops this code from becoming hardware in most systems, that allow only SW-HW invocation.

2.2.3.4 *Hardware Generation and Optimizations*

In this section the most common compiler optimizations for HLS-compiler are explained. While many optimizations are useful (that is profitable) for all kind of (HLS-)compilers, some are dependent on the micro architecture that the compiler creates for the RCU. The existing compilers can be categorized in three groups of micro architectures:

1. The first group uses a processor that gets programmed in microcode as micro architecture. Usually, this contains also a unit or combination of units that are tailored to the compiled program. For example, this can be an ALU with problem specific bitwidths, vector units with many elements, specific Very Large Instruction Word (VLIW) cores, or more complex units (like FFTs-kernel) that are selected from the input application.

2. The second group generates a static scheduled data path. This can be easily generated from a DFG. This only works when the program contains no control flow. When control flow is supported, this is either handled by removing the control flow before when possible (see below), or by using a FSMs. In the later case each *Basic Block* is translated into a separate DFG, with the FSM orchestrating their activation.

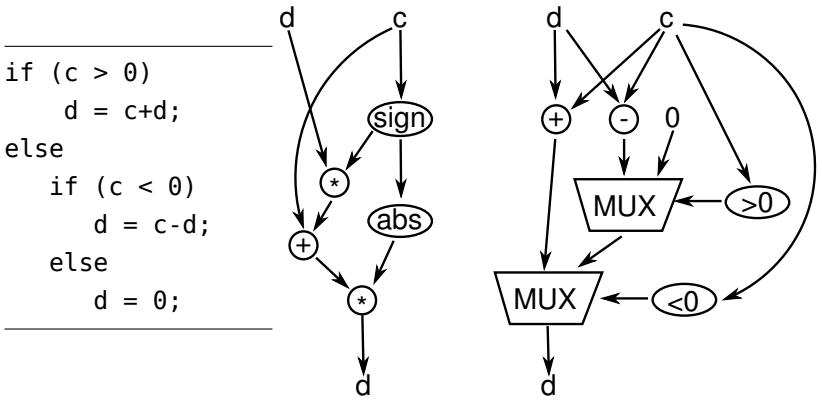


Figure 2.21: Control flow for computing the expression of the variable `d` can be transformed into a data path. This can happen by inserting multiplexers or by mathematical reformulation of the computation into `d=abs(sign(c))*(c+sign(c)*d)`.

3. The third group uses a dynamic runtime scheduling for the generated data path. They are not activated by a static schedule, but dynamically. Each of the operators is activated at run time via signals, often called *tokens*. The term token came from the models of computation these group implements, which are often graphs or networks of operators where tokens flow along the edges and activate the operations. Two of the most often-used models of computation are Petri-Nets ([138, 150]) and Kahn-Process-Networks (KPN) [105, 145]). This scheduling also works with CDFGs and tokens can be used to model the control flow.

The first group can be treated and translated like a CPU. It is actually not really a C-to-HDL compiler, but more a custom-CPU-selector/configurator and a cross-compiler. Therefore, the rest of the discussion excludes these and only includes the last two groups.

REMOVING CONTROL FLOW To reduce the complexity of handling control flow, it can in some cases be reduced to data flow. This is possible when branching leads just to different computations of the same expression and the value can be selected with an inserted Multiplexer (MUX) or by reformulating the expression. Figure 2.21 shows an example of C code containing control flow that can be turned into data flow.

INLINING Having no instructions, data paths have no way to model the software concept of a procedure or function call. So the first transformation that must be performed is in almost all HLS-compilers *Inlining*. In this transformation calls to functions and procedures just get replaced by their definition.

The drawback of the optimization is that it can lead to massive code duplication. While in software this would "only" waste memory and have a negative impact on the instruction cache, when generating spatially distributed hardware this could waste huge amounts of logic resources/area.

This transformation is not able to handle recursion (if it cannot removed by other means, such as tail recursion which can transformed into a loop) and function pointers (if they cannot be removed by other means, such as proofing, that they always call the same function). Thus almost all HDL compilers are unable to handle recursion and function pointers(see Appendix B).

ALIAS ANALYSIS Not an optimization by itself, this analysis is helpful for many other optimizations. One very big problem in many languages is the ability for the programmer to reference and dereference memory addresses directly, usually these references are called *pointers*. C has the operators & and * for this, and allows computation on the memory addresses with arbitrary mathematical expressions. Even the array accesses via [] are translated into those address computations. While it is helpful when the developer needs control over the actual memory layout (for example it could be a low level device driver), it has the drawback that routines can make no assumptions to which address the pointer actually points. Figure 2.22 shows two examples, where the same memory location is possibly accessed via different variables or pointers. The first is a C function copying the contents from one array to another, which can fail when both arrays overlap. The other examples shows a code excerpt, where one of two variables is accessed via pointer. In both cases the alias analysis could eventually detect the potential access to the same memory location via different variables/pointers.

The problem is that when the compiler cannot statically prove that two memory accesses refer to different addresses, much more additional logic is necessary to assure the correctness. In the `cpy` example from Figure 2.22 the copy operation of all `length` elements could be performed in parallel if and only if the compiler can prove that the memory regions do not overlap.

```
void cpy(int *dest, int *src, int
    length) {
  for (int i=0; i<length; i++) {
    dest[i] = src[i];
  }
```

```
int a;
int b;
int *p = &a;
⋮
if (condition)
   p=&b;
⋮
...*p...
```

Figure 2.22: Two examples where alias analysis can detect potential accesses to the same memory location via different pointers.

```
sum = 0;
for (int i=0; i<128; i
    ++)
   sum += array[i];
return sum;
```

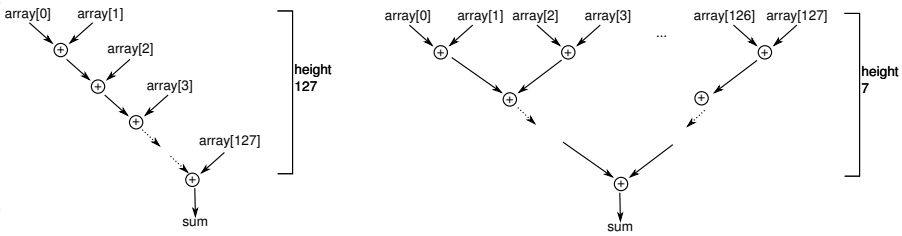


Figure 2.23: An unrolled loop adding up all elements of an array, and the corresponding expression tree (which becomes later the DFG), once without (left) and once with tree height reduction (right).

Should they overlap, it must be assured, that the copy operations of the overlapping parts happen in program order.

In case the compiler could prove that they always reference the same location, one of the references could be optimized away.

Commonly used alias analyses are the points-to analysis from Steensgard [171] and from Shapiro and Horowitz [165]. The result of these analyses is a relation that decides for each pair of variables / accesses whether they must alias, they cannot alias, or they may alias.

TREE HEIGHT REDUCTION A simple optimization that flattens the height of an expression tree. When generating full spatial hardware for this, the tree height correlates directly to the latency. As a consequence a tree with a lower height has a smaller latency with same resource usage. Figure 2.23 gives an example with a loop, that is used to sum up all elements of an array. Without the reduction, the height is equal to the $\#elements - 1$. With the reduction it can reduced to $log_2(\#elements)$. Assuming that each addition takes one cycle, the total latency differs in 120 cycles.

LOOP UNROLLING Unrolling of a loop allows the parallel execution of different iterations of the loop. The tree height reduction example in Figure 2.23 is also the result of such an unrolled loop. In case of no loop carried dependencies this approach is very simple. Should single iterations depend on previous iterations more sophisticated approaches are necessary. One way to improve the loop unrolling is to perform loop transformations, often with the help of a polyhedral framework [109, 154, 185] and modulo scheduling [116]. These allow to unroll loops partially, combine loops, split loops, and change the iteration limits, to remove or reduce the dependency and allow at least partial unrolling ([109, 116, 154, 163, 185]).

PIPELINING A standard technique in hardware generation where additional registers are used to decrease the maximum delay and increase the clock rate. Usually (the exact result depends on the actual numbers of old/new clock rate and required cycles of the computation) this leads to a higher throughput at the cost of a higher latency. The higher throughput is not gained through the (usually slightly) higher clock rate, but due to the fact, that the additional registers allow the start of the next computation before the previous one is finished. A fully pipelined operator has as many operations in computation as it has register stages within. This optimization can be applied on different levels, automatically by synthesis tool that tries to reach a certain clock rate, by the HLS tool that selects pipelined operators, or it can even be supported by the CPU compiler, which transforms loops so they become streaming inputs/output data paths with no loop carried dependencies, which can profit from the high bandwidth pipeline design later.

CHAINING The inverse of Pipelining. Hardware operators in RTL have a registered output. When two operators in sequence have a small combinational delay that, combined would not decrease the clock rate, the registers between them can be removed without sacrificing throughput. Instead the latency is lowered and less resources are required.

This parallelism increasing optimizations are not *always-better* optimization. As the high degree of parallelism and fully spatial designs require correspondingly more hardware area/logic resources it can improve one aspect, while leading to worse solutions with respect to other aspects.

3

PRIOR WORK

History repeats itself. Historians repeat each other.

PHILIP GUEDALLA, Writer

3.1 EVOLUTION OF HW-COMPILERS

In their survey, Martin and Smith divide [129] the history of commercial HLS tools in four generations:

GENERATION 0 (PREHISTORY) The beginning in the 70s, where groundbreaking research was done that laid the foundation for HLS.

GENERATION 1 From 1980 till the early 1990s the first HLS generation consisted mainly out of research products.

GENERATION 2 Starting in the mid-1990s up to the early 2000s most of the major EDA companies, like Synopsys, Cadence and Mentor Graphics, start to offer commercial HLS tools.

GENERATION 3 This is the first generation that is commercially successful (see Figure 3.1). Reasons for this success are manifold. FPGAs, which can be reconfigured and allow for short development cycles became bigger, better and more common. Also the the tools focus on aspects and use dedicated input languages where they can shine: Data flow and DSP applications. With a high demand of those applications for signal and multimedia processing these tools became more and more popular and useful.

So translating abstract high level programs into hardware has a long history, beginning in the 1970s. Over time the tool flow, the libraries and the user interfaces were hugely improved. Though the result quality improved, the compilers still have the same problems today when using a common language, as *C*, as HLL input.

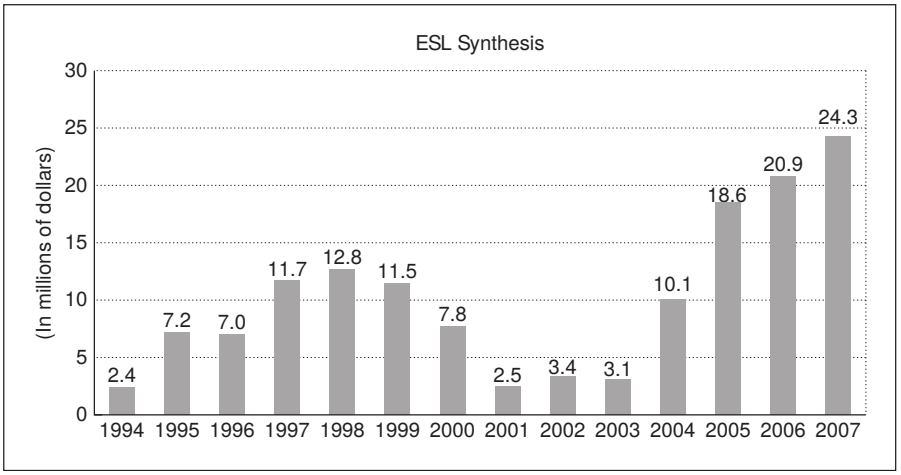


Figure 3.1: Sales of electronic system-level synthesis tools [129].

Synthesizing high level code is handled well by almost all current compilers, as long as the program can be mapped to a pure static scheduled data flow model, which has not much changed since the beginning [57, 58]. Typical commercial approaches for these use cases are MathWorks Simulink HDL-Coder [130] which can generate VHDL/Verilog hardware descriptions, or National Instruments LabVIEW FPGA modules [101].

When using existing software languages, the most widespread approach is to restrict an existing HLL to a certain language subset, forbidding unsupported constructs or types (see Appendix B for the restrictions of C-to-HDL-compilers). Almost all approaches use *C* as source language – as here the legacy code base is very great, and *C* is already closer to hardware than other languages (like e.g. *Java*) – but others exist. As *C* allows arbitrary memory accesses via pointers, many compilers have restrictions regarding them to reduce the necessary overhead when dealing with them (see Section 2.2.3.4). Some forbid the use of pointers entirely (e.g. Appendices B.16, B.17, B.19 and B.22), some just forbid arbitrary references and allow array references (e.g. Appendices B.2, B.10 and B.28), and few allow for arbitrary references, dealing with complex handling for memory accesses (e.g. Appendices B.3 and B.15). Some have more complex restrictions regarding this, such as the *GAUT* compiler (Appendix B.24), which limit the use of pointers to exactly one read and one write access per loop.

Some compilers have restrictions regarding the supported data type; most commonly floating formats are not supported.

Far more restrictions come from the control flow in software. Simple computation is mapped via DFG into a data flow model of computation. Function calls are handled by *inlining* (see Section 2.2.3.4), but as they cannot handle function pointers and recursion, the compilers just forbid them. The rest of the control can be handled but how it is done depends on the compiler.

Using just static scheduled data flow combined with streaming, no control flow logic is allowed and these compilers avoid or limit it by restricting the input language to a subset and/or modifying it, so only certain High Level Language (HLL) constructs are allowed. Examples for such applications are the commercial Impulse-C [99], Handel-C [24], or the academic Transmogrifier C [71]. But in general, static schedules for the data flow do not prevent the compiler from supporting control flow. One solution would be to use a data path for each *Basic Block* of the CFG, and switch between those data paths with a state machine (for example Appendix B.1). Another problem that scheduling has to deal with, are operators with variable length latency. Most often memory accesses fall in to this category. Commonly, they are statically scheduled using the minimum latency of the variable length operator, and in case it takes longer, the rest of the data path is halted.

A different approach uses a dynamic scheduled data flow. It appeared first in Arvind's tagged data flow machine [13]. The dynamic scheduling activates the operators with *tokens*, starting them when necessary. Using these *tokens* not only for temporal activation as a scheduling method, but also for causal activation, this dynamic method implements control flow. The disadvantage of this scheduling is the overhead of the additional token control logic each hardware operator needs. So when using this type of scheduling, the compiler does not only use it to allow easy handling of control flow, but because there are other advantages. For example, *COMRADE* [107] uses the token mechanism to allow speculative execution of operations and speed up the computation; *Pegasus* [26] requires the tokens, as the model of computation of the generated hardware is partially asynchronous, and the tokens are required for synchronization (the authors call it "locally-synchronous, globally asynchronous"). None of the current compilers of the big vendors uses the dynamic approach, all rely on the static approach.

Due to above problems with generic HLL, not only for hardware generation, but for all tasks that are similarly complex, domain specific languages (DSL) are becoming more and more

popular. Research using them for hardware generation has just begun, but is giving already promising results. Examples for such domain specific compilers are *Glacier* [137], which is a hardware compiler for database queries, and *Gaalop* [164] which can generate Verilog for geometric algebra computations.

With using GPGPUs for data processing, multi-core and multi-threading GPU, and vector instruction extensions, the trend in the recent years is to use libraries, languages and language extensions that utilize these highly parallel devices (like CUDA, OpenMP, OpenCL). Using these languages/extension as a source for the HLS can be considered a step between the translation of a generic language and an entire domain specific language. It still allows the formulation of generic problems in HLL, but as the language targets a highly parallel computation model, this can be more easily translated into a (also highly parallel) dataflow model.

A more detailed overview of HDL compilers in general, including the history of hardware compilers is given in [45, 129]; a full survey of hardware compilers targeting C follows in Section 3.2.

3.2 SURVEY OF C-TO-HDL-COMPILERS

As the aim of this work is to improve the compilation of existing high level C code into hardware, many C-to-HDL compilers were looked at and, whenever possible, tried out. The result is this overview of existing technologies and approaches.

The list does not claim to be exhaustive, but should contain at least the more widely used compilers that use C as source language. It is, to the author's knowledge, the most comprehensible overview available. A broader, but not as detailed, survey of HDL in general can be found in [55]. The opposite approach has been taken [131]. In this work Meeus et al. select (depending on relevance and license availability) just a few HLS tools and study them in great detail. Another overview trying to categorize some C-to-HDL-Compilers and some other HDL-Compilers is found in [38].

Also not included in this list are the HDL-Compilers that claim to translate *everything* from a high level language to hardware. They reach this goal by not starting with the abstract high level source code. Instead, they use the generated binary (as input. While this indeed allows for a great degree of freedom regarding the accepted input, several optimizations are no longer possible,

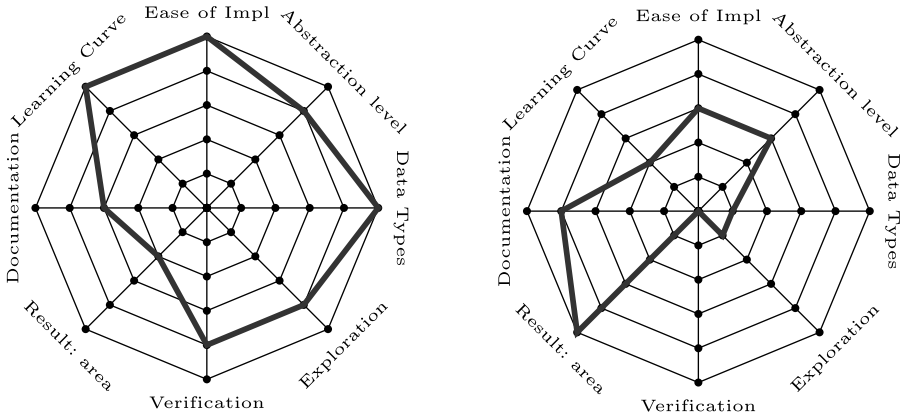


Figure 3.2: Radar charts for the AccelDSP and BlueSpec HLS tools from [131].

and other require complex solutions. One such a compiler is the FREEDOM compiler, presented in [201].

The list is missing many details from many compilers, as most commercial manufactures treat the internals of their software as a trade secret. Direct comparison is not only difficult due to different targets (e.g. synchronous vs. asynchronous hardware), but also due to legal terms. The license often forbids benchmarking software against competitors versions. In [12], Arcas-Abella et al. at least compared four different HLS empirically (not all of them are C-to-HDL).

3.2.1 *Typology*

In [131] Meeus et al. tried to find a set of criteria, that are, from a designers perspective, desired in a HLS tool. From experience they selected the following features, each with a scale from 1 to 5: Ease of implementation, abstraction level, data types, exploration, verification, resulting area, documentation, and learning curve.

As each aspect is represented on a 1 to 5 scale, this allows for easy comparison with radar charts; some of them are shown in Figure 3.2. The authors, Meeus et al., tried to apply an objective scale, but some remain subjective, Table 3.1 gives an overview of the rating scheme used. While the criteria are good for evaluation from a programmer's perspective, they are not as good when looking for a research compiler. There the mode of operation, kind of generated hardware, generated (micro) architecture, and scheduling are more important.

Criteria	1	3	5
Ease of implementation	Several tries necessary before the tool accepting the code	a number of modifications or proprietary language required	original code run with no or minimal modifications
Abstraction level	block level design	explicit timing in the high-level source code	untimed code can be easily scheduled
Data types	floating and fixed point support	conversion blocks for fixed point	no support for floating or fixed point
Exploration	limited exploration capabilities	exploration by swapping predefined blocks	extensive intuitive exploration from script or GUI
Verification	no verification support	source verification no RTL	generates easy-to-use testbenches
Resulting area (example design on Xilinx Virtex2Pro/Virtex 5)	$>$ 3000 slices	...	$<$ 300 slices
Documentation	very limited, hard to understand	...	extensive documentation, easy to understand and explains the tool concepts
Learning curve	steep learning curve	...	flat learning curve

Table 3.1: Criterias used for HLS evaluation in [131], and how they are rated.

So the proposed scheme for classification is oriented at the important steps of the hardware generation and consists of the following aspects: Type of scheduling (dynamic or static), execution and computation model and the generated micro-architecture, when supposed to be coexecuted on hardware and software, how it is partitioned, which language restrictions/extensions are introduced, and what is the target hardware. Also it is interesting if the compiler is an academic project or if it is a commercial product. Academic projects can be more easily used by other research groups, while commercial products are always very closed. Obtaining information about concepts/algorithms used is already difficult, not to mention actual source code to work with.

Some of them are orthogonal to each others (such as product category and partitioning) while other are closely related to each other (for example scheduling, computation model, and execution model). For some of the categories just some keywords for classification are used, the rest is outlined in the description to the compiler. The following keywords for classification are used:

DYNAMIC This is used when the compiler uses a dynamic scheduling approach. Operators activation time is determined dynamically at execution time. The opposite is **static** scheduling.

STATIC Opposite of **dynamic**. Is used when the compiler generates hardware where the operators' activation time is already determined at compile-time.

HW The compiler just generates a hardware kernel. This could be used for anything, from computation kernel in an ACS to some part of an ASIC, but the user has to manually implement the interfacing logic/software (mostly) himself, or some documented protocol is used. Its opposite is **HWSW**.

HWSW The compiler is used to generate hardware and software interface code for HW-SW-coexecution. It generates the hardware and the necessary invocations in the software part. As the software has to be partitioned into parts for HW and SW, the keywords **manual** or **automatic** can also be used to indicate the partitioning method. Its opposite is **HW**.

MANUAL Partitioning necessary for HW-SW-coexecution is performed by the user. Either in form of pragmas, comments in the code, or by explicitly naming the function names the developer has to tell the compiler which part should become hardware, and which stay in software. **Automatic** is the opposite keyword.

AUTOMATIC The HW-SW-coexecution partitioning is performed by the compiler in an automatic fashion (usually profile based). It is opposite of **manual** marked compilers.

ACADEMIC The compiler is an academic work. Even when it is opposite to **commercial**, a compiler can have both keywords, as often the compiler started as academic work and was later continued as a commercial product.

COMMERCIAL The compiler is a commercial product. As the project can have started as an **academic** work, both keywords can be present.

For comparison all the compilers are listed in this format:

NAME OF THE COMPILER

OTHER NAMES / RELATED COMPILERS: Other names used or derived versions of the compiler.

CLASSIFICATION: Classification keywords

AUTHORS / COMPANY: Authors or company who have developed the compiler. Year of the first release.

REFERENCES: References to the compiler (papers, documentations, or press releases).

TARGET: Output, target platform and/or technology.

DESCRIPTION: Short description and further details of the compiler.

RESTRICTIONS: Restrictions of the compiler/parsed input language. When not indicated otherwise all compilers do NOT support recursion, calling of external library

functions, or software callbacks. Also when not otherwise mentioned arbitrary pointers are not allowed. This is due to the fact that the variables must be assigned/tagged in which memory they should be placed. When doing so, the pointer references are hardwired to a a specific memory (bank/type) that is accessed. At run-time a reference to a pointer pointing to a different memory could not be resolved.

3.2.2 *Overview*

The complete list is shown in Appendix B, but just some selected aspects are discussed here. Despite the fact that many of the compilers use C as input, not all of them can be really considered as C-to-HDL compilers. Many use the *C* syntax and require the programmer to code according to a certain a rule set; thus in the end, the programmer is just coding in another HDL, just with *C* (like) syntax.

They are still useful tools, as they often decrease the development time for hardware developers due to their higher abstraction level and often included tools for generation interfaces, test benches, and/or verification models. But they neither allow software programmers with only poor knowledge of the underlying hardware to use it for hardware generation, nor do they allow the fast conversion of legacy code.

It is remarkable that only very few use dynamic scheduling (excluding that compilers that use a processor, that is just *Pegasus* (Appendix B.9), *CHiMPS* (Appendix B.12), *COMRADE 1.0/2.0* (Appendix B.15), *PNgen* (Appendix B.18), *XPP-VC* (Appendix B.40), and *Compaan* (Appendix B.41), which are less than 15%); most rely on static scheduled data paths.

3.3 COMRADE AND COCOMA

The *COMRADE* research compiler was developed at the Department E.I.S of the Technical University of Braunschweig, and later continued at the ESA Group of the Technical University of Darmstadt. As the new compiler generator, described in this monograph, should be capable to create at least a backend, whose (C-to-HDL) functionality matches *COMRADEs*, this section introduces *COMRADE* in more detail than in the survey Section 3.2.

	NIMBLE	*COMRADE*	*COMRADE 2.0*
Target Hardware	Xilinx XC4000 GARP [87]	generic Verilog	generic Verilog
Partitioning	innermost loop	profile based	profile based
Scheduling	static	dynamic simple, token based	dynamic complex, token based (see Section 3.3.2, *COCOMA*)
Memory Model		one cache port	multiple cache ports

Table 3.2: Comparison of *NIMBLE*, *COMRADE*, and *COMRADE 2.0*.

Historically *COMRADE* is a descendant of the *GARP CC* [31] and *NIMBLE*[125] compilers. It is capable of translating innermost loops of C-programs into hardware for *XC4000 Virtex* FPGAs or for the *GARP* architecture ([31]).

3.3.1 *COMRADE*

3.3.1.1 *COMRADE 1.0*

The first *COMRADE*, described in [107], was developed from scratch, using the experiences gained in the development of *NIMBLE*. Its original name was not *COMRADE 1.0*, but just *COMRADE*. The name *COMRADE 1.0* was introduced internally with the introduction of *COMRADE 2.0* to distinguish the two compilers.

COMRADE was built using the *SUIF2* (Stanford University Intermediate Format) compiler infrastructure [3]. The goal was to address all the shortcomings of the *NIMBLE* project, namely the restriction to only innermost loops and the limitation to the already outdated hardware targets. Which means, that *COMRADE* should not only translate arbitrary C-programs (and not just innermost loops) but also should target arbitrary reconfigurable logic.

It implements different passes and uses several tools for help, the relevant steps for the C-to-HDL translation are:

PROFILING The profiling passes for the partitioning and inlining gather execution statistics by block profiling.

PARTITIONING Based on the profiling information the program is partitioned into regions which should execute in

software, and regions which should execute in hardware. The regions that are selected for hardware implementation are checked as to whether they can be actually implemented, or whether they contain operations, that can not be implemented (e.g. *COMRADE* does not support floating point operations). As a very advanced feature *COMRADE* does not reject those as candidates for hardware generation that are unsuitable due to containing code unsuitable for hardware generation, as other compilers do, but instead had the ability to insert software-service calls. These calls allow a stop of the hardware, while making a service request to the software on the CPU. Before and after the request, contents of live variables get exchanged, and after the software service request the hardware continues. For example, a use case for this scenario was code, where just in case of an exception a `printf` was placed inside an `if` to output an error message. While the computation could be accelerated very well, this one not-hardware implementable system call could forbid the hardware generation for normal C-to-HDL compilers, *COMRADE* was able to handle them. Figure 2.20 shows such an use case, Figure 3.3 shows the realization.

HW-SW SELECTION In this pass a library call is inserted that should decide at run time of the program whether the accelerated part should actually run on hardware or if an alternative software version should be called. Also the necessary statements for the transfer of the live variables (variables are live at a point, when they got a definition before the point and a use afterwards) from software to hardware and back are inserted. As the functionality of the software is duplicated in the hardware, and the software selection happens at loop boundaries, this pass was called in *COMRADE* the *Loop Duplication* pass.

In *COMRADE* different IRs were used: First the Abstract Syntax Tree (AST). Afterwards this is lowered to a CFG, and finally a Control Data Flow Graph (CDFG) is generated for the hardware. In between, several optimization and support passes for the hardware generation (i.e. iteration space analysis, SSA conversion, partitioning, insertion of HW/SW-transition operations, constant propagation, bitwidth reduction, tree height reduction, scalar replacement, hardware area estimation, software-service call insertion, hardware generation) take place.

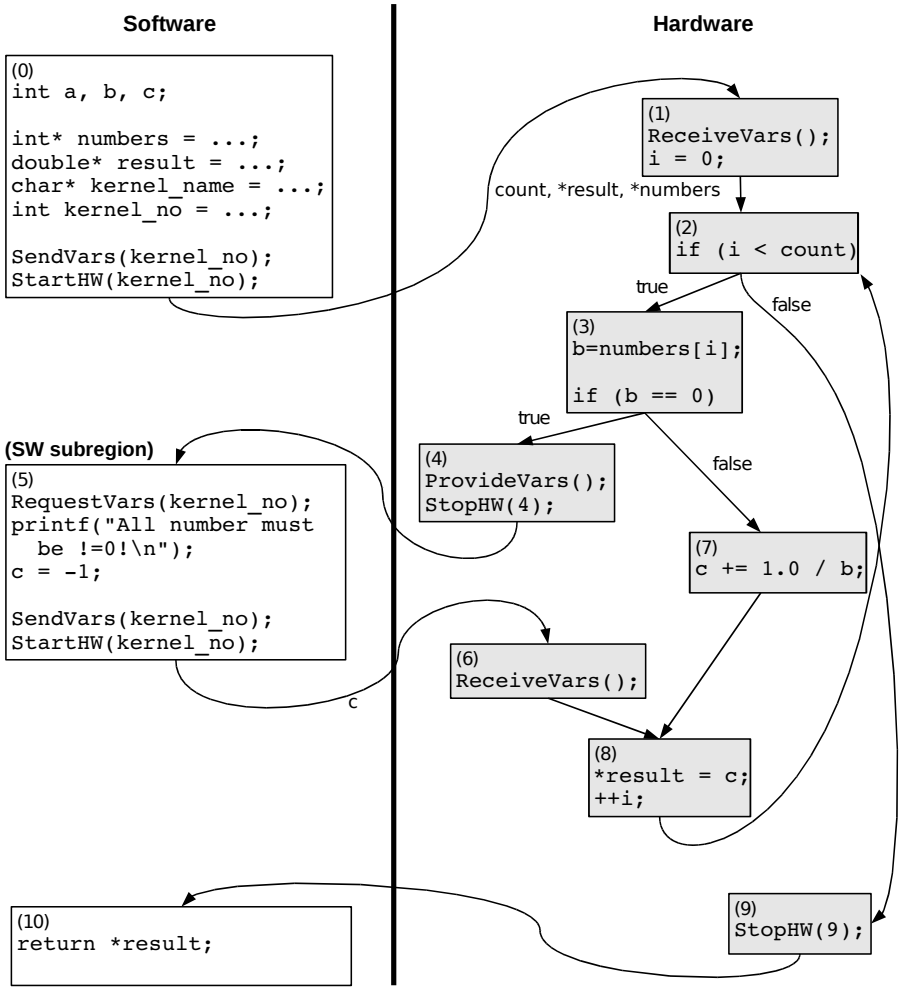


Figure 3.3: *COMRADE* performing a SW service callback for the code from Figure 2.20.

While the original *NIMBLE* had a static scheduler, Kasprzyk proposed a dynamic scheduler for *COMRADE*, using tokens to activate operations. This token based dynamic dataflow goes back to the 1980, to MIT Tagged-Token Dataflow Machine [13] and the Manchester Dataflow Machine [82].

In this runtime scheduling operators are activated when at each input an *Activate* token is available. This signals that the computation responsible for the data at that input is finished. While basic data flow machines only use such activate tokens, *COMRADE* supports speculative and lenient/early evaluation of the operations. This means that for branches the operations in both branch targets could start their computation, even when the branch condition is not yet evaluated. In the branch not taken the computation is canceled as soon as the evaluated condition is available. Figure 3.4 shows how this speculation using *Acti-*

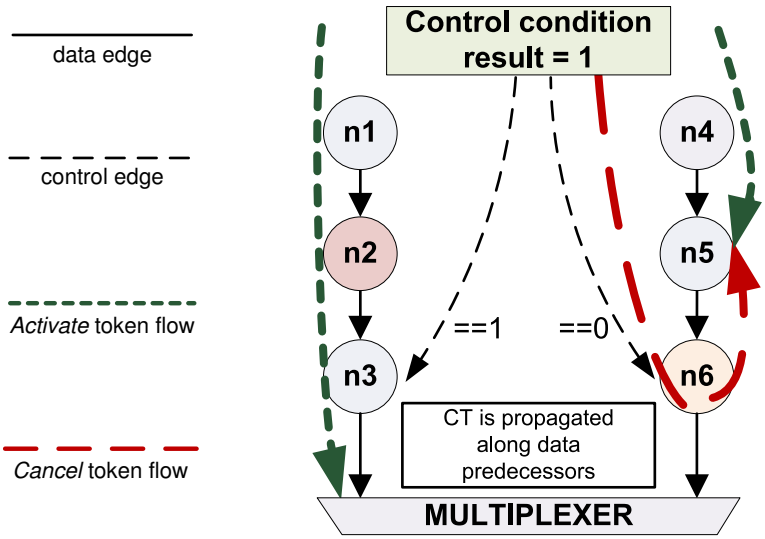


Figure 3.4: Example of *COMRADE*s *Activate* and *Cancel* tokens.

vate and *Cancel* tokens is done (*COMRADE 1.0* called them *Down* and *Up* tokens). In the example both if-branches are speculatively computed, while the condition is evaluated. As soon as the condition computation is finished, a *Cancel* is injected into the not taken branch, that travels upwards to negate *Activate* tokens along it.

3.3.1.2 *COMRADE 1.0 Deficiencies*

While better than *NIMBLE*, *COMRADE* still had deficiencies. Gädke already summarized the problems with *COMRADE* in [67], so just a short overview is given. The first kind of problems are caused by the age of *COMRADE*. The used module library *GLACE* was outdated. Only supporting old technologies (Xilinx XC4000 and Virtex) the included preplacement used in *GLACE* made the generated hardware incompatible with newer devices. Removing the preplacement directives result in wrong meta data, that is used for the area and timing information, which is used for partitioning.

Using *GLACE* also revealed a weakness in the concept/design of the compiler: Neither the used *GLACE* operators nor the generated sequencer supported operator internal pipelining. As result the runtime is increased. Furthermore, the compiler was only designed with integer support. Another problem with the design is that the execution model uses just a single cached memory port to a central cache. This decision originated in the actual used hardware (and thus can also attributed to that): an ACE-V

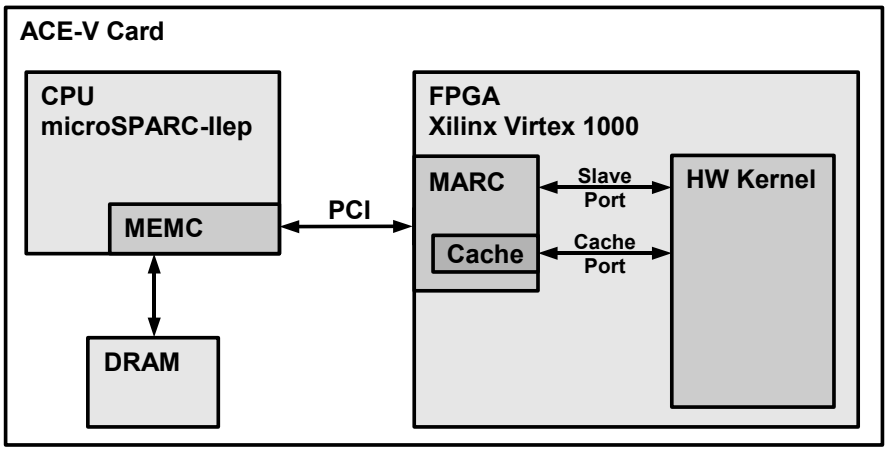


Figure 3.5: The adaptive computer platform ACE-V [67].

[114] adaptive computer that connects a *microSPARC-IIep* CPU and a *Xilinx Virtex 1000* FPGA, both sharing a DRAM main memory accessed via the CPU-integrated memory controller (see Figure 3.5). Using the configurable memory access system *MARC* [117] executed kernels have access to a 32-bit wide memory port with a MARC-internal cache. Even when one neglects the speed advantage which could be gained when using local memories, scratchpad memories, or multiple caches for different memory regions, *COMRADE 1.0* suffered serious problems. For the highest possible acceleration the generated hardware was fully spatial, with no operator sharing. While this is a fast approach, it could not be realized for the memory access operations, as only one port was available. To serialize the accesses a very simple mechanism was used, which inserted memory edges into the CDFG. This should guarantee only one access at a time. But it was so rudimentarily implemented that in certain cases (e.g. parallel data independent and nested loops) the result were again deadlocks or errors in the generated hardware.

The last point is not the only one in the category of problems that are implementation deficiencies. Two areas which were very prone to errors due to its complexity were building the CDFG and the handling of token. Insertion of control edges was incomplete and erroneous. Under certain conditions (e.g. nested loops) no or wrong control edges were generated. This resulted in wrong results or even deadlocks in the generated hardware. Canceling of speculative executed branches was not always correct. In certain cases an operation that should be canceled was not canceled (the operation just forwarded the token, but was

itself not canceled). Here the result could be also deadlocks and wrong results in the generated hardware.

Also in this group of problems is the plain not implemented part of deciding whether a part should be run in hardware or software. *COMRADE* always generated an equivalent software region for hardware regions. So depending on some circumstances and conditions, the program was supposed to decide at runtime which version to choose: The accelerated hardware version, or the software version. In reality the hardware was always chosen (`if (1) …`).

3.3.1.3 COMRADE 2.0

Even when then name *COMRADE 2.0* suggests a completely new rewrite of the old *COMRADE 1.0* it is more of an extension and correction.

The main difference is a new mico-architecture, named *COCOMA* (see next section) and the use of a more modern module library, named *modlib* (see Section 3.3.3). Figure 3.6 shows the compile flow and modules of the compiler.

3.3.2 *COCOMA*

The heart of the new micro-architecture is the COMRADE Controller Micro-Architecture (COCOMA). It encompasses the internal IR used for compilation and the token based computation model derived from it. Similar to the *COMRADE 1.0* approach the hardware is scheduled dynamically with tokens.

3.3.3 *Module Libraries*

A module library (or operator library) collects hardware modules for certain operations. This encapsulates the functionality, so one module implementation can easily be replaced with a different. Some even have different implementations for the same operation, each optimized for other aspects (for example latency, f_{max}, area, power, pipelining capability).

For the compiler developer this is very convenient, as he can rely on such a module library, and does not have to take care of the hardware realization of the operators itself and so does *COMRADE* utilize one as well. It uses a module library that supports simulation and synthesis for all low-level operators that usually appear in C-code (and for all types of data possible).

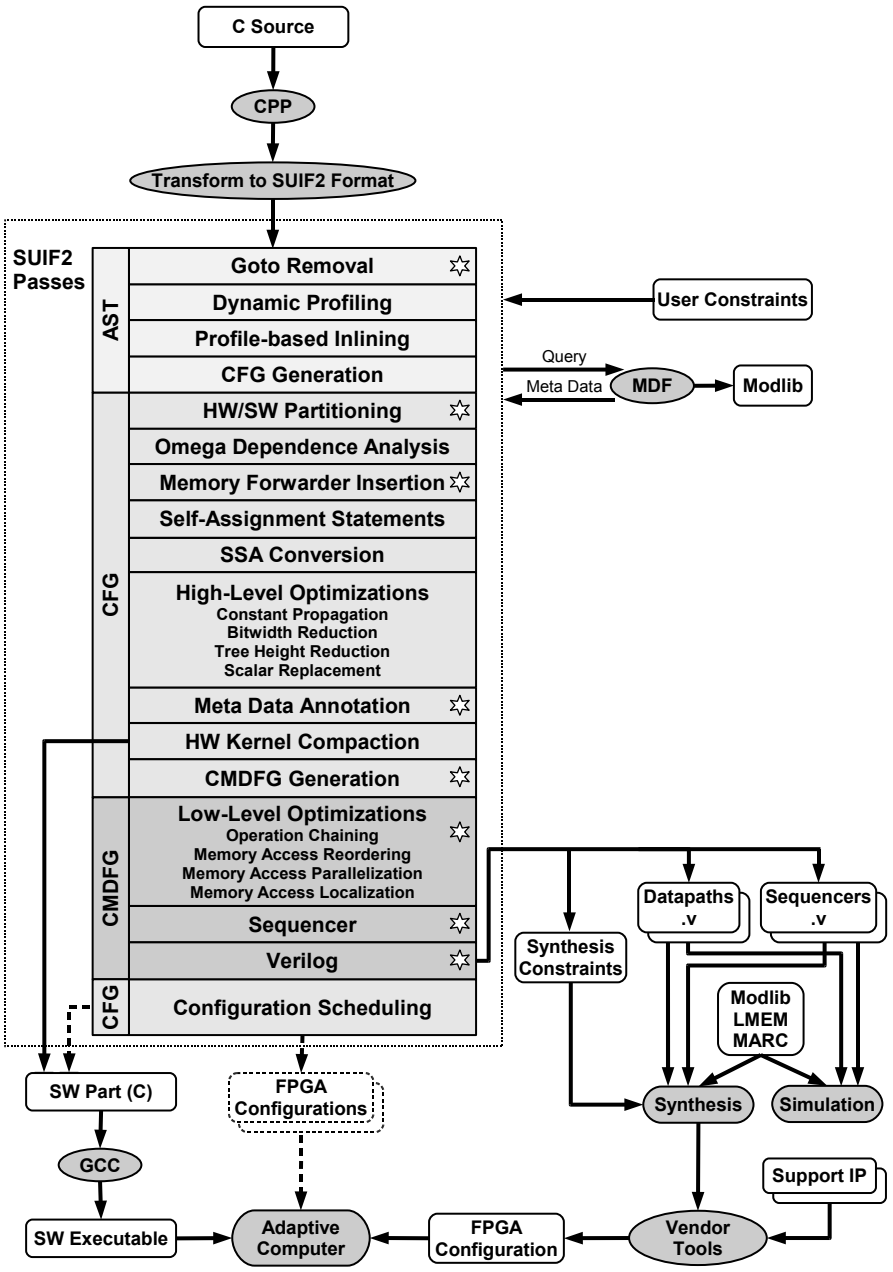
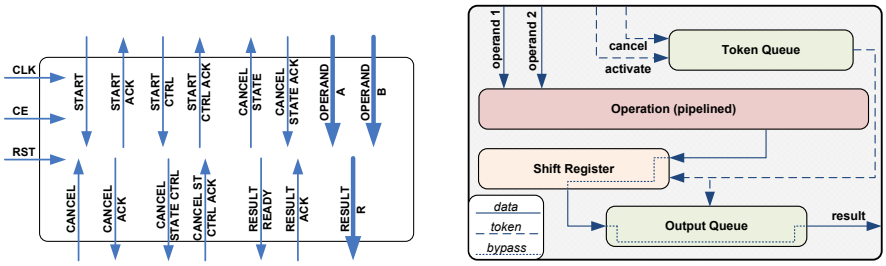


Figure 3.6: The *COMRADE 2.0* compile flow (from [67]).

COMRADE 1.0 uses the module library *GLACE* [140]. It also supplied an interface to get further meta data for the operators, such as required area and latency for an implementation. *GLACE* had the problem, that it only targeted old technologies. Removing preplacement information embedded in the modules to prevent the technology dependency invalidated the meta data, and so *COMRADE 2.0* required a new library.

The *modlib* [180] (short for module library) was created to address the weaknesses of the old operator library *GLACE* and



(a) *modlib* interface.

(b) Internal structure of the *modlib* wrapper.

Figure 3.7: *modlib* module library modules [180].

meet the requirements of *COMRADE 2.0*. The functionality was implemented by either using the operator implementation of the synthesis tool, or by using the Xilinx Coregen IP generator for more complex operators (such as floating point operations). Around the operator itself was the necessary token handling logic for *COMRADE 2.0* wrapped.

Due to its simplicity it integrates well in new projects (and existing, too). So it was used in a couple of other projects ([66, 97, 180]), such as module library for other (DSL-, HLL) compilers and as test bed for new memory systems.

The operations are implemented on the Register Transfer Logic (RTL)-level, so they can easily target different technologies. Modern synthesis tools optimize sufficient well to efficiently handle the low level optimizations and make gate-level designs not necessary. To support *COCOMA*, the operators all have a unique interface that allows one to specify not only the relevant operations for the operators (e.g. signedness and bitwidth) but also to add *FIFO* queues of parametric length at the in- and outputs, as well as parameterized *COCOMA* specific token-handling logic to them. Table 3.3 shows the parameters of such a modlib operator and Figure 3.7 the internal structure and the interface for it.

However, *modlib* lacks a feature which was extensively used with *GLACE*: Area and timing estimation. To compensate for this, a support tool was developed. The resulting *Meta Data Fetcher* (MDF) instantiates the requested *modlib*-operator (with all its used parameters) on the target architecture, and performs the necessary synthesis and place & route steps, to get estimations for area and timing. The estimate is very rough, as timing depends on the critical path, which can be completely different in a small design where no routing congestion happens

Parameters	Meaning
`WA, WB, WR`	Bitwidth of in- and outputs
`Sign`	Signedness
`Depth, QDepth`	Buffer sizes for outgoing data
`TQDepth`	Buffer size for incoming tokens
`StaticCT, PseudoAT` `NoCT, StartCtrlIn`	Flags to control the token handling of the operation
`AreaNSpeed`	Select variant optimized for speed or for area

Table 3.3: Possible general parameters for *modlib* modules. Certain kind of modules (memory and multiplexers) have additional parameters.

compared to a big kernel, where almost no routing resources are available. The same applies to the area, as in later designs the synthesis may select other implementations (e.g. DSP-based multiplication vs. LUT-based) or certain operations can be optimized away in a big kernel and do not require any area at all, or register duplication can increase the area.

3.3.3.1 *COMRADE 2.0 Deficiencies*

COMRADE 2.0, being an improvement over *COMRADE 1.0*, still has a couple of problems. Working with all generations of *COMRADE*, several key difficulties that originate in the framework were recognized by all developers involved:

- The underlying framework *SUIF2* caused a lot of problems. When originally designed, its goal was to act as an infrastructure providing all kinds of optimizations and tools to develop a compiler. However, the project was discontinued shortly just after a level of basic functionality was reached. As result *COMRADE* suffers from missing many optimizations present in other compilers. Comparisons of generated hardware versus the software version was always in favor of the software, as widely-used software compilers often had far better optimizations in their compile flow.
- Related to the first point is the maintainability. *SUIF2* was very ambitious, and to give the compiler developer the

highest possible degree of freedom, it was more-or-less *over-engineered*. To give an impression of the complexity, Figure 3.8 shows a class diagram of the involved classes for *COMRADE*, that are required for just a part of a single pass (in this case it is the *COMRADE* path that annotates the operators with the area and timing information). *SUIF2* even used an own macro preprocessor, which generated certain C-files (so called *hoof*-files). Changes in those files often lead to changes in central header-*includes*, resulting in huge turnaround times while testing and developing.

- Debugging was very complicated due to the different involved languages. Of course, *C* and Verilog always appear in a C-to-Verilog-Compiler, but with the *SUIF2-Hoof*-files and as some steps are implemented in Java (e.g. the query of the *GLACE* library), a total of four languages are involved. Furthermore, the C++ code in the *SUIF2* framework was old. So old that the C++ Standard Template Library (STL) was not mature enough to be considered for use. The authors instead implemented many data structures - which are available nowadays - themselves. While these work, they are incompatible with the STL and incompatible with many modern libraries, which makes it difficult to work with.

- *COMRADE 2.0* was a product of many researchers and students, and as result the code is distributed over many modules, that make up many different compiler passes (necessary *COMRADE 2.0* passes are listed in Table 3.4). When using the compiler, often certain flags have to be set, or changes in the benchmark code were necessary to work around existing bugs. Especially bad was the code responsible for the micro architecture generation, as parts of it were coded in the compiler, and other parts were implemented in the *modlib* , with each relying on the functionality of the other.

Some of the deficiencies mentioned above were addressed: In certain cases, missing optimization could be worked around by manual modification of the original C-code.

To ease debugging several tools were utilized. For visualization of the hardware, the *graphviz* ([62]) software package was used. It reads graph descriptions in a language called *dot* and

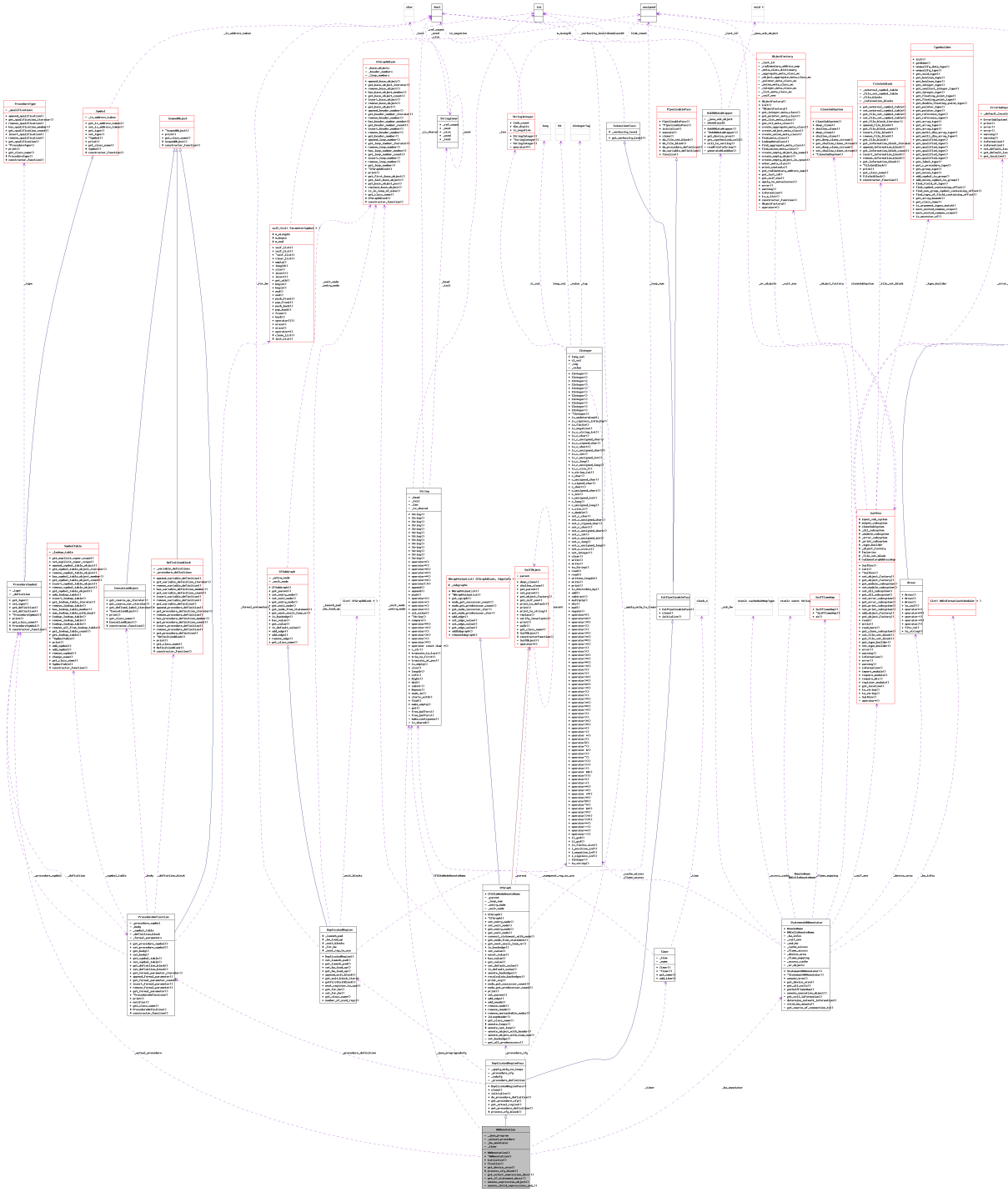


Figure 3.8: Class diagram of the hardware annotation pass (it is intentionally so small, as it should give just an impression of the complexity of the SUIF2 framework).

xcfg_build
ast_passes
dynamic_profiling
cfg_transforms
dismantle_loop
profile_based_inlining
cfg_cloner
cut_unfeasible_nodes
hw_annotation
tree_height_reduction
comrade_naf
comrade_ssa
loop_duplication_pass
decide_hw_sw
dfg_scheduler
cocoma_scheduler

Table 3.4: Necessary *COMRADE* compiler passes.

generates and layouts it in a picture format. This way the generated hardware could be somewhat visualized and debugged.

To assist the debugging even more, especially the dynamic flow of the tokens and the behavior of the hardware, several custom visualization programs were developed. *VeriDebug* [22], *GAP* [4], and *pnnsGraph* [148] all operate on text output generated while simulating the hardware. The generated Verilog code contains nonsyntheziable output instructions printing cycle and token information of the operators. They visualize the hardware similar to *graphviz* output, but now add the additional dimension of time. They allow for the replay of the dynamic hardware, going backwards and forwards. Another drawback of the generated static *dot* graph was the size. It often grows very big, making it difficult to find the relevant operators. The above programs helped here, too, by allowing to search for certain operators, restrict the nodes shown to selected parts of the graph, and to hide all kinds of edges (i.e. control, memory, data edges). Figures 3.9 to 3.11 show screenshots of the programs while debugging a graph of *COMRADE* generated hardware.

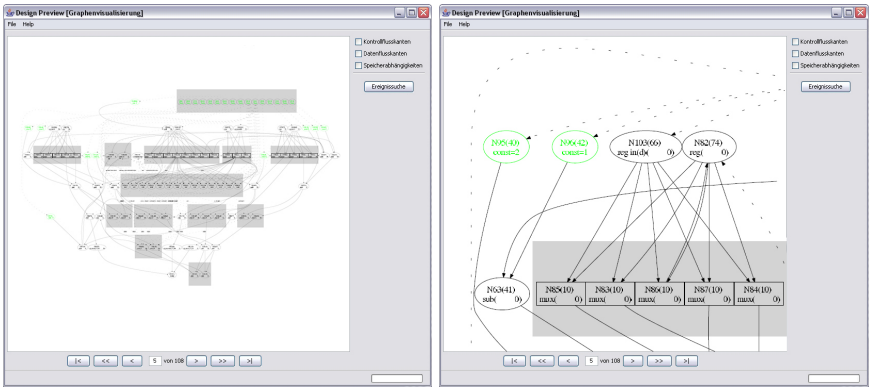


Figure 3.9: Screenshot of the program *GAP* debugging *COMRADE* generated hardware.

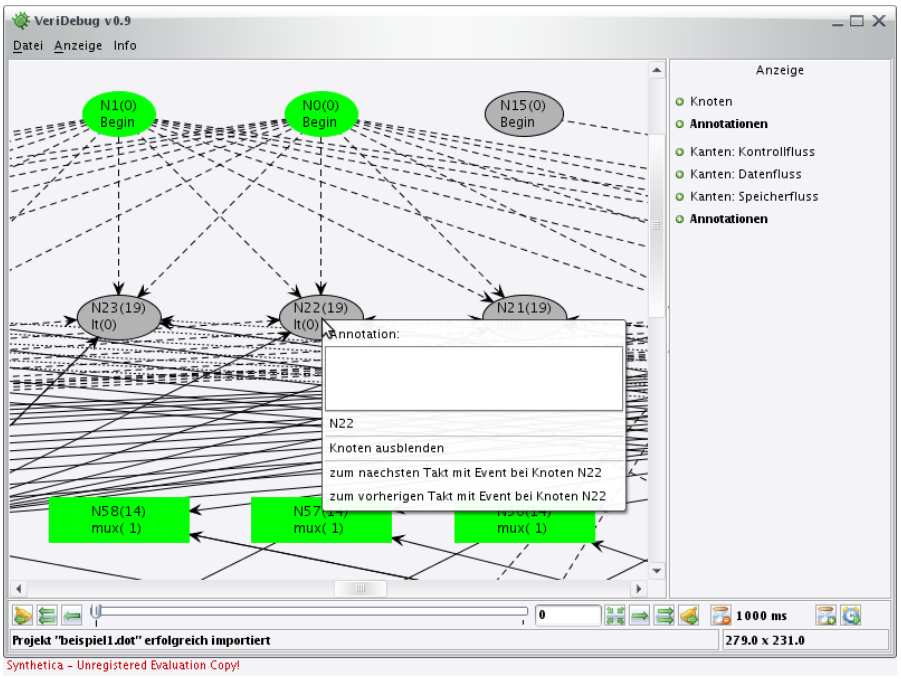


Figure 3.10: Screenshot of the program *VeriDebug* debugging *COMRADE* generated hardware.

3.4 THE *SCALE* COMPILER FRAMEWORK

To overcome the problems with the *SUIF2* framework, a new compiler framework for a reimplementation was sought. The requirements for the new framework were:

1. It should provide SSA-representation and alias analysis besides basic loop optimizations.

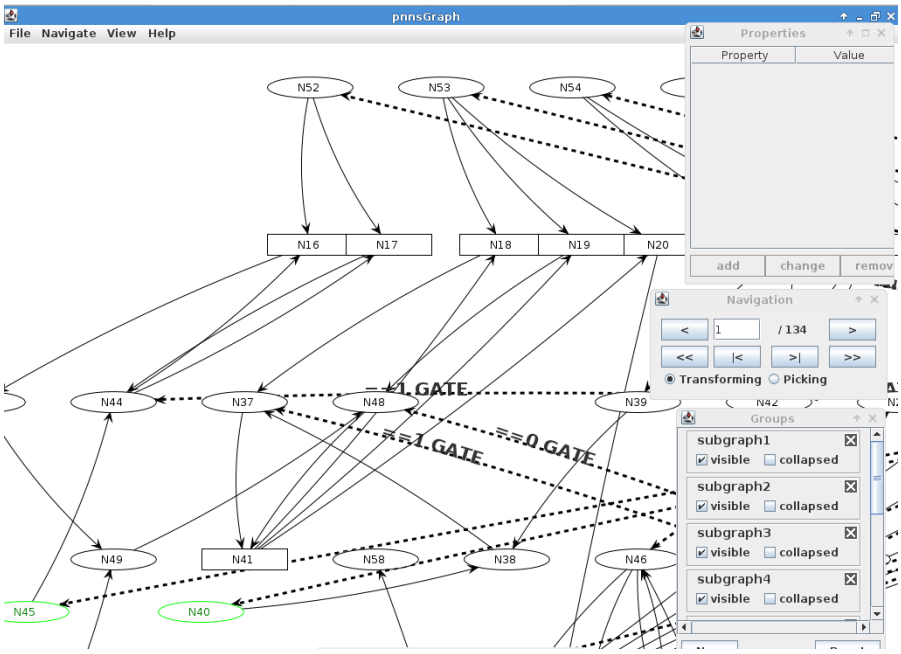


Figure 3.11: Screenshot of the program *pnnsGraph* debugging *COMRADE* generated hardware.

2. It should be extensible and easy to maintain. So it should provide access to all necessary data structures and use a single, versatile IR.

3. It should be programmed in Java. As the curriculum of the university does not contain C, this decision was made to involve more students.

There are not many compilers fulfilling Item 1 and Item 3, and the most common choices were filtered out (like gcc[170] or llvm[121]) due to the fact, that they were written in C/C++. Only the COmpiler INfraStructure (COINS) [41, 162] and the Scalable Compiler for Analytical Experiments (Scale) [163, 167] meet all of these requirements. While more sophisticated and having more optimizations than *Scale* COINS fails in Item 2. It has a number of different IRs depending on the level of abstraction (HIR and LIR), which make Scale the only remaining candidate.

Scale was originally developed as research compiler for the TRIPS-architecture[27], but also supports other backends (Alpha, PowerPC, Sparc or C) and frontends (Fortran). The framework supports

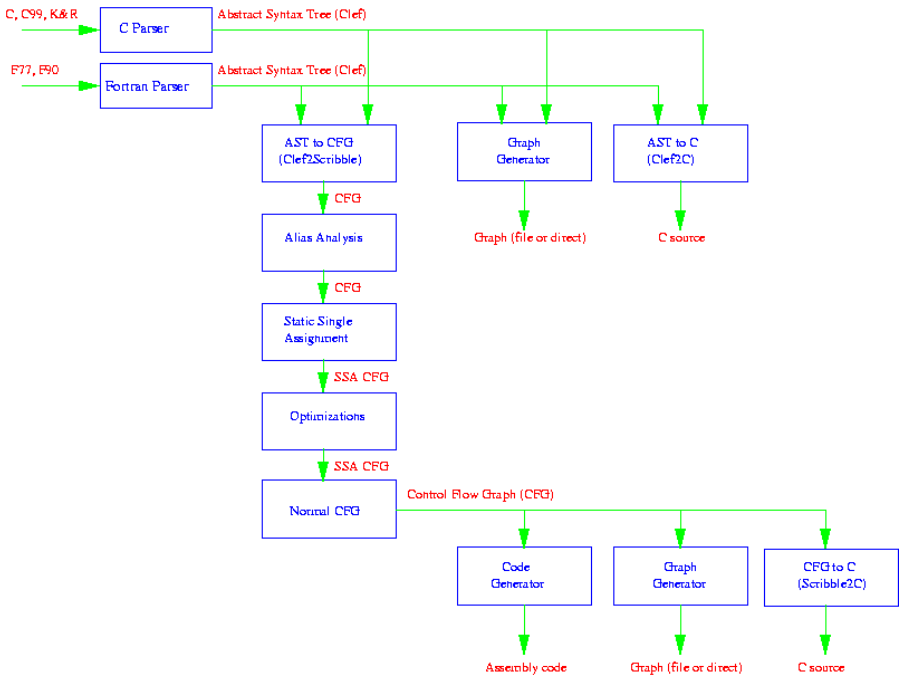


Figure 3.12: Data flow of the Scale compiler (from [163]).

- Inlining
- Alias Analysis
- Copy Propagation
- Value Numbering
- Loop Invariant Code Motion
- Loop Un-rolling
- SSA-form (and automatic conversion into and out-of it)
- Sparse Conditional Constant Propagation
- Loop Transformations (Loop interchange, tiling, distribution, unrolling, and fusion, and strip mining)
- Scalar Replacement for Array Elements
- Partial Redundancy Elimination
- Global Variable Replacement
- Useless Copy Removal
- Dead Variable and Code Elimination
- Basic Block Redundant Load and Store Elimination
- Expression Tree Height Reduction
- Converting implicit loops to explicit loops
- Converting irreducible graphs to reducible graphs
- Annotations of CFG- and AST-nodes

The flow of the compiler (see also Figure 3.12) follows the standard sequence: Parsing the program, building an AST, converting into a CFG (called Scribble), apply to the AST and CFG different optimizations, and emit in the back end machine code (or C).

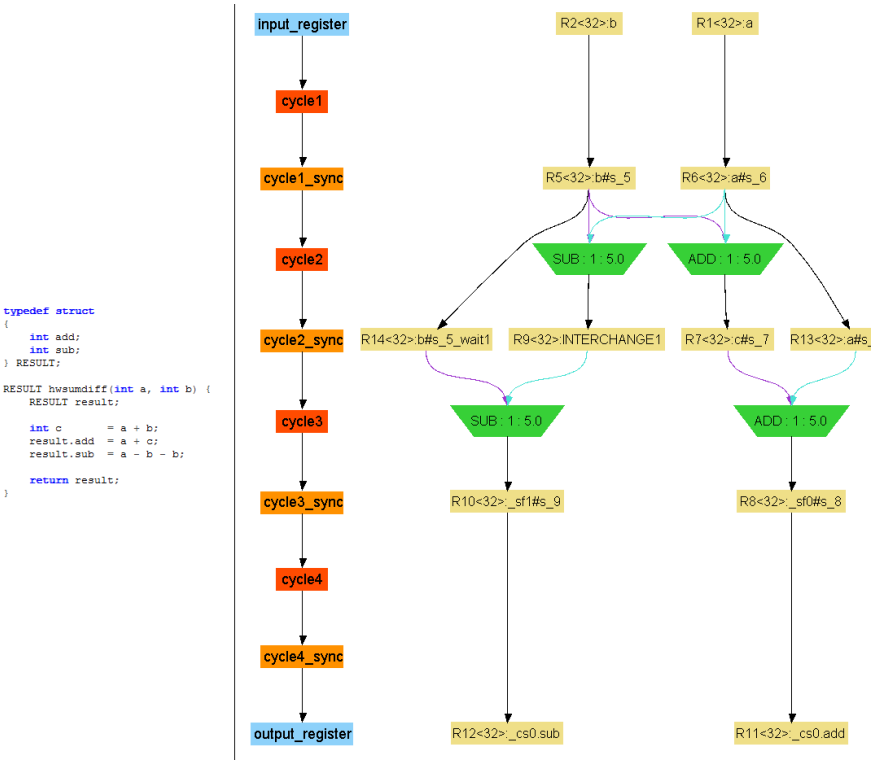


Figure 3.13: Example for hardware generation with the *hardScale* backend.

3.4.1 hardScale *Compiler*

As first hardware generating backend for the *Scale* compiler, *hardScale* was implemented at the ESA Group of the TU Darmstadt. It was extended with a HW-SW-Partitioning algorithm that allowed *hardScale* [94] to automatically partition the compiled program into parts for HW-SW-coexecution. This is similar to *COMRADE* but it uses a more sophisticated whole program path profiling algorithm instead of a block profiling.

Also callbacks from the generated hardware to the software (called software service) were possible.

In addition to the automatic partitioning manual partition with *pragma* instructions is possible. While more versatile than *COMRADE* in this regard, the generated hardware is just statically scheduled and had several other restrictions: Jumps and accesses to global variables were forbidden and all pointers must use the *restrict* keyword. Figure 3.13 shows an example C-to-HDL translation with *hardScale*.

In the rest of this work, the name *hardScale* is used when referring to the generated compiler as whole. When specific IRs

or internals are used, the name *Scale* is used, to indicate that the used parts belong to the original *Scale*.

4

C-TO-HDL-COMPILER GENERATOR *TRIAD*

I'd rather have a search engine or a compiler on a deserted island than a game.

JOHN CARMACK,
Programmer

As already stated in the previous chapters, the aim of this work is to create a compiler which is capable of generating a C-to-HDL compiler that synthesizes dynamically scheduled hardware. Using a versatile description as input, it should be possible to easily implement different scheduling schemes.

Triad[1] is not only the name of the compiler but also the name for the file type and input description language to the *Triad* compiler. It is described in the following sections in detail.

The general idea is to modify the *hardScale* compiler from Section 3.4.1, so it can generate Verilog HDL code from generic *C* code for dynamic scheduled hardware. The proposed flow extends the currently used flow around the (statically scheduled) hardware compiler with an additional step. This additional step takes place before hardware synthesis, in order to automatically generate a new compiler *backend* from a compiler description.

Figure 4.1 shows the flow in full detail. Before using the *hardScale* compiler for hardware generation, *Triad* must be run, to generate the necessary files for the hardware backend. *Triad* reads a specification file (whose format is also called *.triad*) containing the hardware modules used and their mapping to *Scale*s IR. Furthermore, the file holds the definition of the implemented scheduler. This definition declares the token types used and rules for token interaction with the operators. As result, *Triad* generates several *.java* files that extend the *hardScale* compiler to implement the hardware generation with the described scheduling scheme. In addition, some Verilog wrapper modules are generated that encapsulate the operators and possible token handling functions. Using these, the *hardScale* compiler can generate dynamically scheduled hardware according to the specified rules.

1 For the curious, the name is chosen to be *in tune* with the *scale* naming scheme

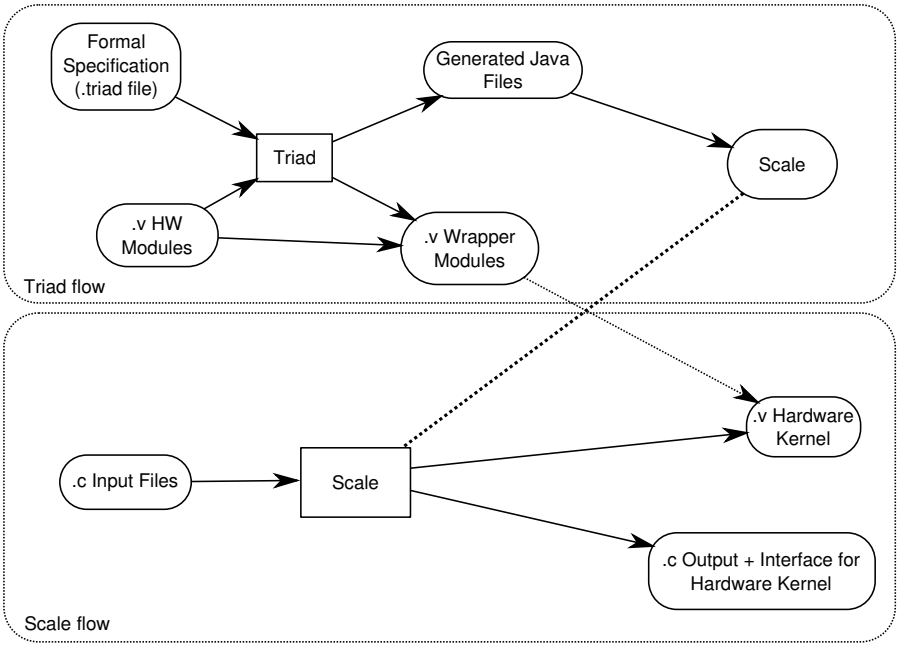


Figure 4.1: Compile flow with *Triad* and *Scale* and how both interact.

To describe when things happen at some point in this flow, the term *generation-time* in addition to the commonly used terms *compile-time* and *run-time* is introduced. While the latter two terms have the common meaning, they refer to the *hardScale* compiler, i.e. *compile-time* refers to time when the *hardScale* compiler compiles the C program to Verilog and *run-time* to the moment when the generated Verilog is executed, *generation-time* refers to the time frame when *Triad* is executed and the *hardScale* hardware synthesis backend is generated.

4.1 IDEAS AND CONCEPTS

As its main and most important feature, `Triad` is intended to generate a C-to-HDL Compiler using token based dynamic scheduling, like *COMRADEs COCOMA* micro-architecture. In [67], Gädke developed a set of mathematical rules, that formally describe the behavior of *COCOMA*. Actually, these rules were not the formal base for *COCOMA*, but were retroactively laid down to describe the supposed/implemented behavior. They describe the values for expression and signals as results using a kind of first order predicate logic expressions (see Appendix of [67] for the detailed rules). Because of the versatility of predicate logic, and to build on the existing *COCOMA* rules, a similar mathematical approach was used as base for the `Triad` input de-

scription. From these rules a logic controlling the token flow is generated for each operator. This logic is called *token controller*.

Additional information that is required to generate the micro-architecture are the hardware operations and a mapping from the internal abstract *hardScale* operations to the concrete hardware operations.

From this information *Triad* generates the backend for the *hardScale* compiler. Internally, this backend uses a CDFG from which to generate a data path. The scheduling of datapath operations is done at run-time, using the compile-time generated token flow architecture, which in turn was defined at generation-time.

4.2 TRIAD

As mentioned above, *Triad* needs three pieces of information as input: The hardware operators itself, the mapping to *Scale* expressions of these operators, and finally the rules for the scheduling algorithm.

These descriptions are combined in one specification text file (called *.triad*) that is input to *Triad*.

The syntax of the file is given as syntax diagram and Extended Backus-Naur Form (EBNF) [191] in Appendix C, a semantic description follows here.

4.2.1 *Hardware Operators*

The intention is to use hardware operators similar to the operators of the old *modlib* operator library that was already used in earlier HW-compilers. To be more flexible, the modules used for the operators are not hard coded into the compiler. Instead, Verilog modules that are available can be specified, together with their bitwidth and signedness.

Four special module names are reserved: `memread`, `memwrite` for memory accesses, `arrayaccess` for array accesses and, finally, `mux` for control flow. These modules are not mandatory, and can be omitted when the generated compiler is intended to translate only programs without them. Table 4.1 lists these optional modules with their function.

In the specification file, the modules used are specified in the `HWOPS`-section. As can be seen in Figure 4.2, the specification for each operator contains

- its Verilog module name.

module name	required when translated program contains ...	module functionality
`memread`	reading memory accesses	performs a reading memory access
`memwrite`	writing memory accesses	performs a writing memory access
`arrayaccess`	array index computations	computes base + index ∗ size + offset with constant size
`mux`	control flow	2-input-multiplexer

Table 4.1: Optional hardware operations that must be defined in the *Triad* file, in case the translated program requires them.

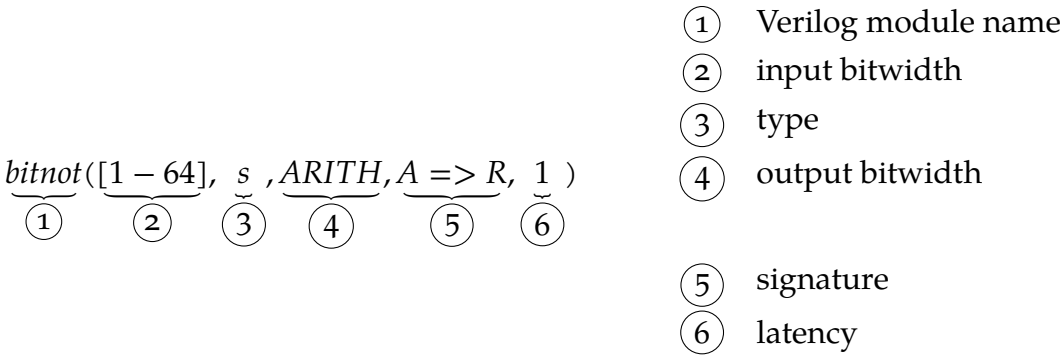


Figure 4.2: *Triad* module definition

- its input bitwidth: As the compiler requires many operators that differ just on the size of the operands and usually Verilog modules are implemented with support of different bitwidths, here all supported bitwidth can be specified. A set notation is used that also supports ranges of numbers.
- its type: Here the type of the input data can be defined with a single letter: Possible are signed (`s`) and unsigned integer (`u`), floating point type (`f`) and a undefined/don't-care type (`x`). The latter one is used for bit operations.
- its output bitwidth: Here only two different output sizes are supported, which depend on the type of function. Arithmetic functions (marked with the `ARITH` keyword) have a result, that has the same size as the inputs, while logical functions (marked with the `LOG` keyword) have an output size of 1.

Listing 4.1: Example of a *Triad* file, HWOPS-section.

```
HWOPS:
       lognot([1-64],s, LOG, A => R, 1)
       bitnot([1-64],s, ARITH, A => R, 1)
       logand([1-64],s, LOG, A*B => r , 1)
       bitand([1-64],x)
       logor([1-64],s, LOG, A*B => R, 1)
       bitor([1-64],x)
       mod([1-64],s)
       subfloat([32,64],f)
       bitxor([1-64],s)

       mul([1-64],s)
       mulfloat([32,64],f)
       addfloat([1-64],f)
       sub([1-64],s)
       divint([1-64],s)
       divfloat([32,64],f)
       add([1-64],s)
       mul([1-64],s)
       div([1-64],s)

       cmplt([1-64],s, LOG, A*B => R, 1)
       cmpgt([1-64],s, LOG, A*B => R, 1)
       cmplteq([1-64],s, LOG, A*B => R, 1)
       cmpgteq([1-64],s, LOG, A*B => R, 1)
       cmpeq([1-64],s, LOG, A*B => R, 1)
       cmpneq([1-64],s, LOG, A*B => R, 1)

       bitshift([1-64],x)
       abs([1-64],s, ARITH, A => R, 1)
       neg([1-64],s, ARITH, A => R, 1)
```

- its signature and argument names: Here the names of the input arguments are assigned to the name of the output argument (names refer to the names in the Verilog code).
- its latency: The latency of the operation in cycles. As operators with variable latency are also supported, the latency can be preceded by a > to indicate variable latency and declare a lower bound.

Just name, bitwidth, and type are mandatory, the rest is optional. They default to the most common values in practice: `(ARITH, A * B => R, 1)`.

An example for the HWOPS-section is shown in Listing 4.1.

$$\underbrace{AdditionExpr}_{①}(\underbrace{SINT, UINT}_{②}:$$
$$\underbrace{add}_{③}\ \underbrace{(WA = \%b, WB = \%b, WR = \%b)}_{④}())$$

① *Scale* expression name
② type limitation
③ module name from `HWOPS`-section
④ instantiation parameters

Figure 4.3: *Triad* expression to hw module mapping

4.2.2 *Operator Mapping*

Scales internal CDFG representation has nodes that contain expressions for all instructions inside them. The different expressions are modeled inside the compiler by means of a class hierarchy. The *expression* class hierarchy is shown in Table 4.2. In the hardware generation process (described in Section 4.3) these expressions get implemented in hardware with the hardware operators from the `HWOPS`-section. Thus, a mapping between the internal *Scale expressions* and the hardware operators should be defined in the *Triad* file. Not all expressions of the hierarchy need to be mapped, only the computational ones, as the others, such as the `PhiExpr`, are handled implicitly by the translation process (see Section 4.3). Even for the computational expressions, none is truly mandatory, the *hardScale* compiler will just not be able to translate programs that contain an unmapped expression (a compile-time error is thrown). Table 4.2 gives a recommendation which expressions should be mapped.

A simple mapping from *Scale expressions* to hardware operators, such as `AdditionExpr => add` is not enough, as an expression can represent different types (e.g. an `AdditionExpr` handles float additions and integer additions). To handle a single *expression* type (of multiple different types) with different hardware modules, it must be possible to constrain the type information in the mapping. Therefore, in the expression-hardware-module-mapping, the mapping can be restricted to selected types. Possible types are *signed integer*, *unsigned integer*, *double*, and *float*. To support modules that can handle different bitwidths, and maybe require other parameters, it is also possible to define Verilog pa-

Listing 4.2: Scale expression to hardware module mapping for addition.

```
AdditionExpr(SINT,UINT : add(WA=%b, WB=%b, WR=%b, SIGN=%s)())
AdditionExpr(FLOAT,DOUBLE : addfloat(WA=%b, WB=%b, WR=%b)())
```

rameters which are necessary for the hardware module instantiation.

In the *Triad* file, the mapping is defined in a dedicated section, the `MAPPING`-section. It consists of multiple lines, where each line defines a mapping. One such mapping is shown in Figure 4.3. Each mapping consists of the *Scale* expression that should be mapped, the types to which this mapping is restricted, and the module name (specified in the `HWOPS`-section), that should be used, together with its instantiation parameters. They can get assigned either constant values or some placeholders, which get replaced at instantiation time. Possible placeholders include:

`%b`: Is replaced by the input bitwidth of the operation. Even when there are two inputs, only one bitwidth is available. According to the *C* standard, operators in *C* operate on operands of the same type. *Scale* handles type (and also size) conversions with explicit expressions.

`%s`: Represents the signedness when handling integers (0 = unsigned, 1 = signed).

As an example, all the mappings defining an addition operation are shown in Listing 4.2. Both commands define how the `AdditionExpr` from *Scale* is mapped. The first mapping restricts the mapping to the signed and unsigned integer data types and instantiates a Verilog module named `add` for the addition, passing the bitwidth and signedness as parameters. The second line of the example maps additions with float and double data types to the `addfloat` Verilog module and uses just the bitwidth as parameter.

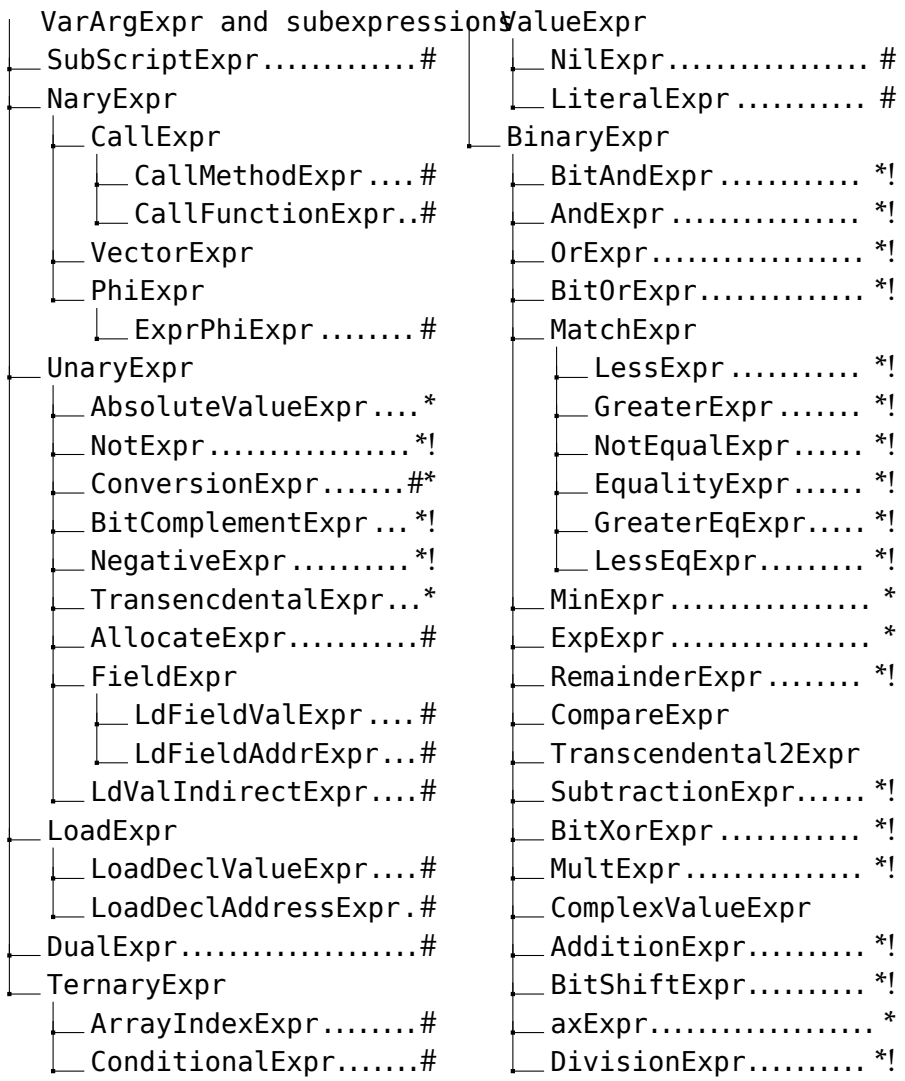


Table 4.2: *Scale* expressions and recommended mapping settings. Only the one marked with * can be defined (addtional ! means they are recommended), marked with a # means they are already handled by some other mechanism, and the rest can be ignored for the hardware translation process.

4.2.3 *Scheduling Rules*

The heart of *Triad* is of course the generation of the dynamic, token-based micro architecture for the scheduler. Simple runtime dynamically scheduled hardware uses tokens that flow with the computed data and start operations when all required inputs are available. More sophisticated approaches use multiple token types; an example of this is *COMRADE*, which uses two different types, one for activating and one for canceling speculated computations.

The positions where the tokens are stored are called *places*. To make the description as versatile as possible, the tokens can be placed at the inputs of the operators and at the outputs. Their names are respectively *input places* and *output places*. These places have no nodes on their own in the generated CDFG. The generated graph uses the mapped operations as nodes. Each node has input and output places associated with it.

For the rules that decide how the token flows from node to node, and how different tokens interact with each other, a mathematical approach was chosen. The required formulation needs to define, for each type of node, at which condition which action is taken. Such formulations are not new and the style used in *Triad* is inspired by the set-builder notation for describing sets and the ECA (Event-Condition-Action)[60] model used in reactive systems. The ECA rules in the systems react to *events* by checking a *condition* and starting an *action*, using predicates to describe the events and conditions. The common form in mathematics to describe a set, is the set-builder notation (like $\{x \in D | predicate(x)\}$), where all elements x from a domain D belong to the set, when the predicate is true. A similar concept is also used in the functional hardware description language *Bluespec* [159], named *Guarded Atomic Action*. Similar to ECA they describe a rule by triggering an action when a logical predicate evaluates to true.

Using the same principle, rules are described by three components with the help of logic expressions, using sets, quantifiers and predicates:

ENTITY SELECTION decides to which set/subset of objects (graph nodes, input places, output places) the rule is applied.

GUARD CONDITION contains a first order predicate logic term, checking for the condition.

Listing 4.3: Example rule in Triad.

```
{Node node|(∃ Inputs(node) input: isGlobal(input))
  ∧ hasToken(input, Cancel)}
=>  delete(input, Cancel);
```

ACTION that is executed, when the guard condition evaluates to true.

To define the rules, the mathematical class notation for sets is used. In general a (simplified) rule looks like this (and an example is in Listing 4.3):

```
{ Entity variable | booleanExpression(variable) } => action ;
  Entity Selection      Guard Condition              Action
```

For clarity, this is a little bit simplified. Actually, the boolean expression can depend on more than one variable, so the entity selection is a list and can contain multiple node types and variables. The boolean expression is a first order term in predicate logic. That means it consists of predicates and quantifiers. For all nodes that have a true evaluation of the expression, the action is triggered. As often more than one action has to be triggered with the same condition, multiple actions can be specified.

In the following section the details of the implementation of the rules description in the *Triad* file are given, followed by an example.

4.2.4 *Token Specification*

The *Triad* file also has a `TOKEN`-section where the tokens for the dynamic scheduling are defined. A set of tokens is defined just as a sequential list with the name of the different token types. As some models have mutually exclusive tokens at an operation (such *activate* and *cancel*) and these can be encoded with viewer resources (see below for details), the tokens in a token set are such a combination of mutually exclusive tokens. When the tokens are not mutually exclusive, multiple token sets can be defined. The design idea behind it is that this allows different underlying token implementations and optimization of the token representation.

Listing 4.4 gives two possible ways of defining a micro-architecture that uses three different token types: `Activate`, `Cancel`, and `ACK`. Each token set is contained in a `()` pair result in different implementations.

Listing 4.4: Two ways to declare three tokens *Activate*, *Cancel* and *ACK*. The first will require 2 bits to represent the 4 states (0 = no token, 1 = *Activate* token, 2 = *Cancel* token, 3 = *ACK* token. The second needs three bits, one for each token. In both cases the *Activate* token is both, start and end token.

```
// one token set, with three mutual exclusive tokens
TOKENS: (*Activate*,Cancel,ACK)

// three token sets, each with just one token
TOKENS: (*Activate*)(Cancel)(ACK)
```

0	0	no token
0	1	Activate token
1	0	Cancel token
1	1	ACK token

(a) Encoding of three mutually exclusive tokens. 2 bits for signaling and storing are enough.

0	0	0	no token
1	?	?	Activate token
?	1	?	Cancel token
?	?	1	ACK token

(b) Encoding of three non mutually exclusive tokens. 3 bits for signaling and storing required.

Figure 4.4: Realization of the different tokens sets from Listing 4.4.

Similar to state machines in hardware design, where different ways of encoding the current state are used, different encodings can be used for the tokens. The most obvious implementation for tokens is to model them just as a signal, with 0 meaning a given token is not present, and 1 meaning it is present. In total the width of the token bus/storage for tokens (signaling/storing all kind of tokens) is as big as the number of tokens. But when the tokens are mutually exclusive, the different tokens can be encoded as a single value (with an additional value indicating the absence of all tokens). Then the resulting token bus/memory has just logarithmic size. Section 4.2.4 shows the implementation of this approach for the example token set.

The optimized encoding is not mandatory, it is still possible (and necessary when the tokens are not mutually exclusive) to enforce the other one-hot encoding, by defining the tokens so that they are not in a single mutual exclusive token set, but each token is in its own token set. The implementation for this approach in the example can be seen in Section 4.2.4.

The generated hardware does not only need data in- and outputs, but also some generic control logic, namely a signaling to start the generated hardware and some way for the datapath to signal the end of the computation. For this purpose one and only one token across all sets must be tagged with a "*" before its name to declare it as start token, and exactly one with a "*" after its name to declare it as end token. Both declarations can be on the same token.

When the hardware is started, at all token places that belong to data inputs a start token is created. And the other way around, when at all output places of a global output node (one that has no successor) an end token is present, the hardware signals that the computation is finished.

4.2.5 *Entity Selection*

The *Entity Selection* part of the rules selects the entities that are checked for the rule. In the same way as in the set-builder notation a variable is bound to a domain (or using programming language terms: a type) a variable name is bound to an entity type. The entities can be nodes of the CDFG, input or output places that belong to the nodes, or token names.

To make the writing of the rules easier, some helper entities were defined, which are just a subset of another kind of entity. These could have been defined with regular entities and a predicate, but formulated that way they would have bloated the rules.

An example would be the helper entity `MemOps`, which describes all nodes of the CDFG that are a memory operation. Instead of using an entity of its own a definition just using the `Node` would have to use an additional `isMemOps()` predicate.

The same applies to the parametric entity sets. The formulation

```
{Node node|(∃ Inputs(node) input: isGlobal(input))
  ∧ hasToken(input, Cancel)}
=>  delete(input, Cancel);
```

is just a short form of

```
{Node node|(∃ Input input: isGlobal(input) ∧
  isInputOf(input, node)) ∧ hasToken(input, Cancel)}
=>  delete(input, Cancel);
```

where the predicate `isInputOf` is replaced by the parametric entity set `Inputsnode` in the quantifier.

Entity	Semantic
`Node`	All nodes in the generated CDFG graph.
`HWOps`	All nodes that perform computation (i.e. no control flow managing nodes).
`MemOps`	All nodes that perform memory operations (i.e. reads and writes).
`Input`	All input places (of all nodes).
`Output`	All output places (of all nodes).
`InOut`	All in- and output places (of all nodes).
`Token`	All token names
`Inputs(`*`node`*`)`	All input places of a certain node `node`.
`Outputs(`*`node`*`)`	All output places of a certain node `node`.
`Succs(`*`node`*`)`	All successor nodes of a certain node `node`.
`Succs(`*`output`*`)`	All successor places of a certain output place `output`.
`Preds(`*`node`*`)`	All predecessor nodes of a certain node `node`.
`Preds(`*`input`*`)`	All (i.e. only one) predecessor places of a certain input place `input`.

Table 4.3: Entities in *Triad*. The types, which are allowed only in the predicate, are easily identified by the fact that they are parametric.

These parametric entity sets require a bound variable and can therefore not be used for the entity selection. They can only be used in the guarded condition terms.

Table 4.3 lists all entities.

4.2.6 *Guard Condition*

The *guard condition* is a mathematical first order logic predicate. Logic functions, like the logic *and*, *or* or *not* can be used, as well as the universal and existential quantifiers and predefined predicates to form boolean expressions (which trigger the actions).

Predicate	evaluates to true :⇔
`hasToken(e, tokenName)`	At entity `e` a token of type `tokenName`
`isSucc(node1, node2)`	Node `node1` is direct successor of node `node2`

Predicate	evaluates to true :⇔
`isSucc(node1, node2, edgeType)`	Node `node1` is direct successor of node `node2` via an edge of type `edgeType`
`isPred(node1, node2)`	Node `node1` is direct predecessor of node `node2`
`isPred(node1, node2, edgeType)`	Node `node1` is direct predecessor of node `node2` via an edge of type `edgeType`
`isFinished(node)`	Node `node` has finished its computation
`isCycle(integer)`	The cycle counter has reached (exactly) `integer`
`isControlledBy(node1, node2)`	Node `node1` controls Node `node2` (means `node1` is connected to the select input of a multiplexer node, which has on one of its inputs paths `node2`, on the other not).
`isTokenType(token, tokenName)`	Token `token` is of the type `tokenName`.
`isGlobal(io)`	The input `io` is an input from outside the datapath into it/the output `io` is a result.
`isDominator(node1, node2)`	Node `node1` is a dominator of node `node2`
`isPostDominator(node1, node2)`	Node `node1` is a post dominator of node `node2`
`isConnected(input, output)`	The input `input` is directly connected to output `output`.
`isInputOf(input, node)`	The input `input` is input of the node `node`.
`isOutputOf(output, node)`	The output `output` is output of the node `node`.
`isImmedMemoryPred( node1, node2)`	In the serialized memory chain Node `node1` is immediate predecessor of `node2`.

Predicate	evaluates to true :⇔
`isControlledAndTrue( entity, node)`	If entity `entity` is controlled by node `node` and currently the controlling condition selects branch containing `entity`. to
`isControlledAndFalse( entity, node)`	If entity `entity` is controlled by node `node` and currently the controlling condition does not select branch containing `entity`.
`isJoinNode(node)`	If node `node` is a joining node (i.e. it is a multiplexer).
`isConditionNode(node)`	If node `node` is a node whose output is controlling control flow.
`isSelectInput(input)`	If place `input` belongs to a select input of a multiplexer.
`isLoopInitMux(node)`	If node `node` is a multiplexer handling the initialization of loop variables.
`isInitInput(input)`	If input `input` is the init input for a multiplexer node handling the initialization of loop variables.
`isMemOps(node)`	If node `node` is a memory operation
`isRunning(node)`	If node `node` is performing a computation.
`isConstant(node)`	If node `node` is a constant node.

Table 4.4: Predicates in *Triad*.

Table 4.4 lists the available predicates. These include functionality to check for kind of nodes (for example memory node, global input or output node) or properties of a node. These properties include dynamic properties, which can only evaluated at run-time, like if input or output places hold certain tokens, or static properties, which can be evaluated at compile-time, like structure properties, such as predecessor, successor nodes, or if a node is (post-) dominated by another node.

4.2.7 *Action*

When the guard condition predicate evaluates to true, the listed actions are executed. Table 4.5 gives a list of the predefined ac-

Action function	Semantic
`create(e, token)`	creates a token of type *token* at given entity place *e*
`delete(e, token)`	deletes a token of type *token* at given entity place *e*
`start(node)`	start the operation of the node *node*
`reset(node)`	resets (cancels) the operation of node *node*

Table 4.5: Actions in *Triad*.

Listing 4.5: Example rule in *Triad*. It checks for a certain condition between two nodes `node1` and `node2`, and creates and deletes tokens at them if the condition becomes true.

```
{ Node node1, Node node2 | ∃ Outputs(node1) output, Inputs(node2)
     input:
                            allSuccsWithoutToken(output) ∧
                                hasToken(output, Activate) }
                          => delete(output, Activate),
                             create(input, Activate);
```

tions that can be triggered. These actions include creation and deletion of tokens at input and output places, as well as the starting and aborting of an operation.

In the action the variables from the entity selection and from the quantifiers can be used. The action is triggered for all possible parameter combinations, even when one of the variables is bound in an existential quantifier where only one value evaluates to true. To illustrate this behavior, we take a look at the example in Listing 4.3. Just the fact that one input from `node` is a global input is enough to evaluate the predicate to true, but the create action would be invoked for *all* inputs *i* that would make that predicate true.

Listing 4.6: Example macro definition in the RULES-section of *Triad*. When used, just a textual substitution places the output argument in the expanded version.

```
allSuccsWithoutToken(output) := ∀ Succs(output) ins:
                                    !hasToken(ins, Activate);
```

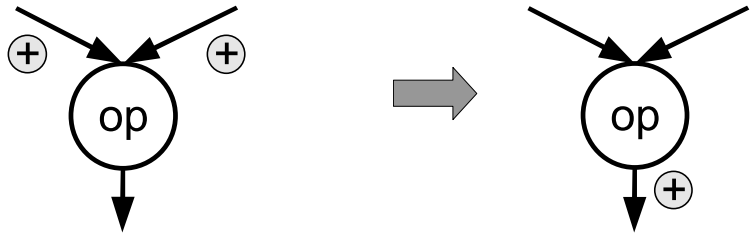
4.2.8 *Macros*

To ease the description of the rules, a macro feature was implemented for the rules section. Hence, frequently used boolean expressions need not be repeated in all rules in which they appear. Instead, a custom predicate can be defined as a macro, and later be reused. As it is just a convenience function for the implementer, the macro facility is very simple. In the definition of the macro other macros can be used, but their definition must appear before. The formal parameters just get textually replaced by the actual arguments. Therefore, each macro has its own namespace, so used variables can be arbitrarily chosen, regardless of the used context. Listing 4.6 shows such a macro definition.

4.2.9 *Simple Token Based Scheduling*

Here a short minimal example for a rule set is presented, longer examples follow in the next chapter. In the simplest case of a token based schedule an *activate* token flows with the data. The computation of an operator is started when at all inputs of the operator an *activate* token indicates the availability of a data (shown in Figure 4.5a). Upon start the *activate* tokens are consumed, and as soon as the computation of the operator is finished a new token is generated at the output place of the operator (shown in Figure 4.5b). The *Triad* rules describing this behavior are shown in Listing 4.7 (the first rule describes the start of the operation, the second the creation of the token).

The token is then forwarded to the operators that are connected to this operator's output. In Listing 4.7 the macro definition and third rule describe this behavior. To avoid transfering a token to an operator which has not yet started, and still has a token waiting, the rules delay the token forwarding, until all successor places are free to take the next token. This can lead to serious back pressure upwards in the DFG. To handle the back pressure at the node above the waiting node, the first rule has



(a) When all inputs got a token, the operation is started and the tokens are removed.

(b) When the operation is finished, a new token is generated at the output.

Figure 4.5: Token based scheduling with just one *activate* token.

to be extended, so it only starts computation, when the resulting token space is empty. This is simply done by appending

```
∧!(∃ Outputs(node) output:hasToken(output, Activate))
```

to the guard condition of the first rule.

For pure data flow graphs this is sufficient, but if control flow is contained in the CDFG, the required token flow is more complex. What happens is that the control flow splits, and then later joins again (in case of a loop this can also happen the other way around, that the join appears before the split).

At first glance two solutions are possible:

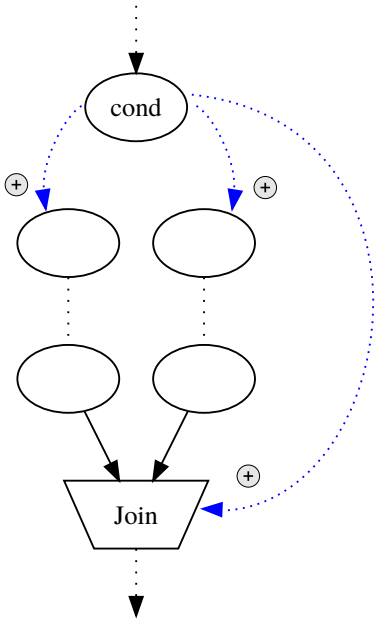
1. The nodes after the split node both get an *activate* token and start the computation. Later, the join node waits for all *activate* tokens at the input places, that is from all computation branches, as well as from the evaluation from the condition. The advantage of this method is that computation on all branches is started before the condition is evaluated, making this speculative execution faster than the other variant (see Figure 4.6a).

   An optimization in this case would be to just wait for the *activate* token from the condition and the *activate* token of the actual taken branch. Depending on the latency of each of the branches, an additional logic to dismiss the token of the other branch can be necessary.

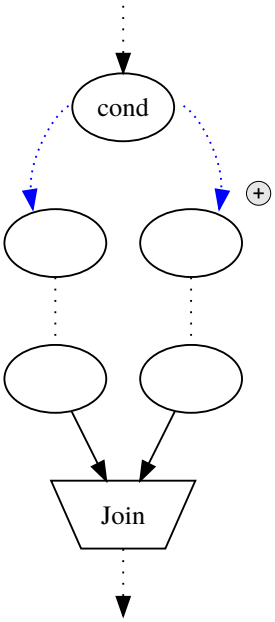
2. The nodes after the split nodes do not all get an *activate* token, but instead wait for an *activate* token from the condition. This means, the join node does not need to be connected to the condition and just waits for an *activate* token at only *one* of its inputs (see Figure 4.6b. The drawback is that the computation of the branch does not start before

Listing 4.7: *Triad* rules section for scheduling pure data flow with an *activate* token.

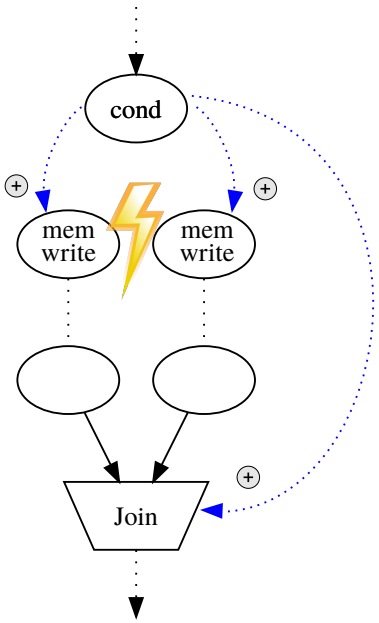
```
RULES:

// when all inputs of an operation have a token, remove the
    tokens and start the operation
{ Node node | ∀( Inputs(node) input: hasToken(input, Activate) )
     }
     =>  start(node),
         delete(input, Activate);

// when an operation is finished, create activate token at
    output
{ Node node, Output o | isFinished(node) ∧  isOutputOf(o, node)}
     =>  create( o, Activate);

// now define the token forwarding
allPredsWithoutToken(output) := ∀ Succs(output) ins: !hasToken(
    ins, Activate);

{ Node node1, Node node2 | ∃ Outputs(node1) output, Inputs(node2
    ) input:
                              allSuccsWithoutToken(output)∧
                               hasToken(output, Activate) }

     => delete(output, Activate),
        create(input, Activate);
```



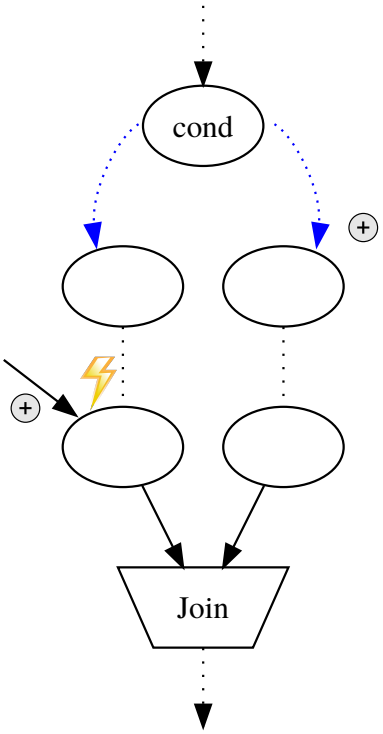
(a) Token from the condition is forwarded to all nodes following a split and to the join node



(b) Token from the condition is only forwarded to the first node(s) in the taken branch.



(c) Token from unconditional operator waits erroneously at operator in non taken branch.



(d) Speculative executed memory write can lead to wrong data.

Figure 4.6: Control flow in a very simple token flow model.

the condition is evaluated, leading to a higher latency of the computation.

Both methods do not work correctly in all cases. Actually some operators are not free of side effects. The set of operators include memory operations, which can lead to wrong computations when executed speculatively (without additional handling) (shown in Figure 4.6d).

The other case has the problem that the operators in the untaken branch can use data from unconditional nodes. These unconditional operators send their *activate* token, but the actual computation does not happen as the node is not starting the computation, because the operation itself is in the untaken branch. In case of a re-execution (for example because this is part of a loop), these already present *activate* tokens represent the data from the previous computation. In case the branch is now taken and the *activate* token in the branch reaches the operator, it instantly starts operation, because it is wrongly activated from the already waiting *activate* token from the previous computation (shown in Figure 4.6c).

In general these problems can be handled by using more tokens and rules. Both problems can be solved by using some kind of *cancel* tokens: In the first case, they would be injected in all input places in the untaken branch that are connected to nodes outside the untaken branch. In the second case the memory access would get a dependency from the condition node by inserting a rule that transfers an *activation* token from the condition to them. *Cancel* tokens flowing in the opposite direction from the multiplexer in the branch not taken would cancel the *activate* tokens flowing downwards. Another approach could be made by using a speculative memory system, such as *Precore* [181].

4.2.10 *Requirements and Limitations*

For the proof-of-concept implementation of *Triad* a number of simplifying assumptions were made. First, only a single memory accessible by a single read/write port is modeled.

Second, the generated compiler allows only synchronous hardware (even though the token based approach works well with asynchronous hardware [26]) implemented in Verilog, and the supported operators must be pipelined and conform to the following interface: While in- and outputs can have arbitrary names (they are specified in the `Triad`-file, in the *HWOPS*-

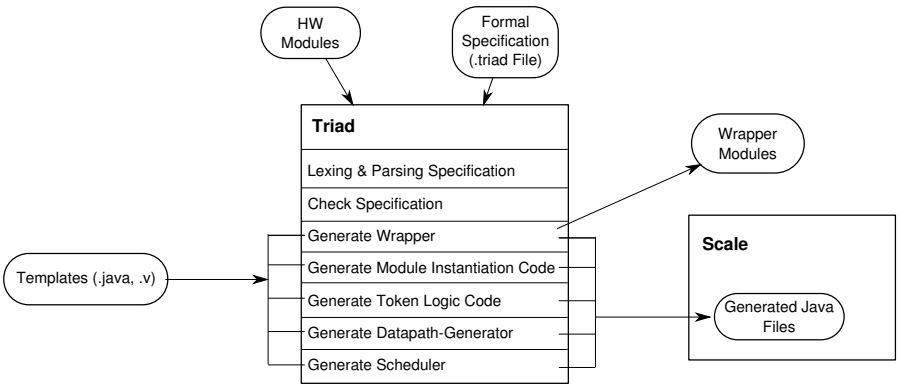


Figure 4.7: Steps during generation of the compiler.

section with the signature), the signal names for the common infrastructure signals are fixed (`CLK, RESET, CE`). For the specific modules that perform the memory access, there are a number of additional signals, like `ADDR, DATA, DATA_RE, DATA_VALID` for reading and `ADDR, DATA, DATA_WE` for writing. Further, each operator with a variable latency needs a `finished` signal.

4.2.11 *Implementation*

The implementation of the *Triad* compiler does not contain complex algorithms, as the most complex decision are done at compile-time and not at generation-time. As a given rule potentially leads to similar or same logic at different nodes it means that a different node could actually have the identical token controller. Thus, their controllers implement the same functionality at different nodes. A possible implementation could try to enumerate all different token controllers that are possible and code them into the backend.

While it would be possible to generate all possible token controllers for the given rules, and just instantiate them at compile-time, this approach has a severe drawback. Considering that each rule selecting a generic node can be responsible (or not) for an operator, that already means that $O(2^{\#\text{rules}})$ rules combinations are possible. This number is also the theoretically reachable upper bound for different token controller types. As this is too much to generate beforehand, the generation is delayed to compile-time, when only the actually necessary token controller get generated.

Figure 4.7 shows the internal flow for *Triad*. First, the *Triad* file is read by an ANTLR [147] built parser, and basic syntax and sanity checking is done.

Next the code for the wrapping and instantiation code for the operator modules is generated and added to the *hardScale* compiler.

Afterwards, the code for the generation of the token controller from the rules is emitted and finally the code which is necessary to build the data path from the scale internal representation is generated.

The generated wrappers simply embed the original operators and a generic token controller module, whose functionality is generated later at compile-time. Listing 4.8 shows such a wrapper and Listing 4.9 shows the generated Java code for the instantiation and bookkeeping.

The code generation is either done by writing the complete code out, or by reading some template .java/.v files and inserting the required missing functionality. The *hardScale* compiler is already modified, so that functionality, that does not differ with different backends, is already pre-implemented in its own base classes. Newly generated Java files are copied into the source tree and extend those base classes and override and/or implement backend specific functionality, so that even without or with broken code generation the rest of the *hardScale* compiler would be runnable.

Details of the generated code are in explained in the next section.

4.3 GENERATED *HARDSCALE* BACKEND

Similar to other systems (for example, *gcc* uses a such an approach where back ends are generated from a machine description file), just the backend is generated, and front and middle ends are reused. The compiler used was *hardScale* from Section 3.4.1.

The generated backend gets invoked from *hardScale* for each region that was selected for hardware generation. Besides the manual partitioning with pragmas (see example in Listing 4.10) and later test bench generation, no other capabilities from *hardScale* were used.

Listing 4.12 shows the pseudo code for the hardware generation. At the beginning the *Scale* CFG (named *Scribble*) in SSA is translated into a CDFG. The SSA forms allow an easy transla-

Listing 4.8: Example of generated wrapper for a Verilog *add* module (excerpt).

```
module add_Triad_Wrapper #(parameter
                           WA = 32,
                           WB = 32,
                           WR = 32,
                           TOKEN_CONTROLLER_LOGIC = 0,
                           TOKEN_SIGNAL_IN_WIDTH  = 1,
                           TOKEN_SIGNAL_OUT_WIDTH = 1
                          )(
                              input  wire [(WA-1):0] A,
                              input  wire [(WB-1):0] B,
                              output wire [(WR-1):0] R,
                              input  wire                                  RESET,
                              input  wire                                  CLK,
                              input  wire                                  CE,
                              input  wire [(TOKEN_SIGNAL_IN_WIDTH-1):0]
                                   TOKEN_SIGNALS_IN,
                              output wire [(TOKEN_SIGNAL_OUT_WIDTH-1):0]
                                   TOKEN_SIGNALS_OUT
                          );

add #(
    .WA(WA),
    .WB(WA),
    .WR(WA)
) original_module (
    .A(A),
    .B(B),
    .R(R),
    .RESET(RESET),
    .CLK(CLK),
    .CE(CE)
);

TokenControlLogic #(
    .VARIANT(TOKEN_CONTROLLER_LOGIC),
    .IN_WIDTH(TOKEN_SIGNAL_IN_WIDTH),
    .OUT_WIDTH(TOKEN_SIGNAL_OUT_WIDTH)
) tcl (
    .TOKEN_SIGNALS_IN(TOKEN_SIGNALS_IN),
    .TOKEN_SIGNALS_OUT(TOKEN_SIGNALS_OUT)
);

endmodule
```

Listing 4.9: Generated Java code for instantiation of an *add* module.

```
/**
 *   Template file for the Scale compiler, that gets modified by Triad backend
      generator.
 *
 * @author Florian Stock
 * @author Triad-Generator
 *
 * Used Parameters are referenced via %N\textdollar\ style where N referes to the
     Nth argument,
 * all types are Strings (i.e. the conversion must be "%Ns" (just one percent, two
      are here needed for escape ;-)).
 * This template is instantiated with the following parameters:
 *   1  Name of the hardware operation
 *   2  Constructor Code
 *   3  Other methods/members
 */
package triad;

class HardwareOperation_add extends DataFlowOperation {

    private static int runningID = 1;

    /**
     * Creates new Instance of this hardware operation, and the name
     * for it is generated from the operation name and a running number.
     */
    HardwareOperation_add() {
        this("add_"+String.format("%05d",runningID++));
    }

    HardwareOperation_add(String name) {
        super(name,"add");
        createInputPort("A");
        createInputPort("B");
        createOutputPort("R");
        latency = 1;
        latencyVariable = false;
    }

    static public int getCount() {
        return runningID-1;
    }

    public String verilogInstantiation() {
        StringBuilder result = new StringBuilder();
        result.append("add_Triad_Wrapper  ");
        result.append(getName());
        result.append(" (\n");
        result.append(getVerilogInstantiationParameters());
        result.append("\n);\n");
        return result.toString();
    }
}
```

Listing 4.10: Example for selecting C code in a program for HLS. Everything between both pragmas is selected.

```
#include <stdlib.h>

int main() {

int a = 0xaffe;
int b = 0xdead;

int rnd = random();
int result;

#pragma hardware on
  rnd = rnd % 4;

  switch (rnd + 1) {
        case 0: result = a+b;
                break;
        case 1: result = a-b;
                break;
        case 2: result = a;
                break;
        case 5: result = b;
                break;
        }

#pragma hardware off
  return result;
}
```

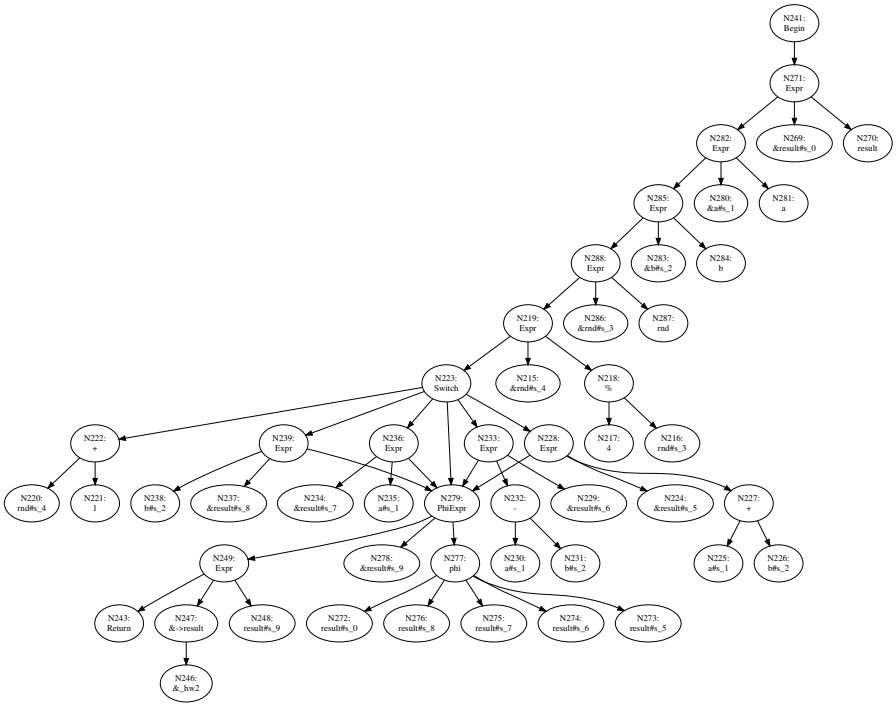


Figure 4.8: Internal *Scale* representation *Scribble* (CFG) of the program from Listing 4.10.

tion of the data flow part, as each piece of data has exactly one definition. In Figure 4.8 the CFG representing the program from Listing 4.10 can be seen.

The first step of the translation is the identification of the loops. These are independently handled by the process in reverse topological order. This assures that inner loops are handled before an outer loop.

Each loop is translated as a standalone datapath, using a virtual operation to represent inner loops in an outer loop containing them. The translation of the nodes itself is done with a visitor pattern. While the loop identification and generation code does not change with the generated compiler, the actual visitor code is generated by *Triad* and overrides a base class visitor. Listing 4.11 shows parts of such a visitor, handling an *add* operation. As a *Scale* CFG expression can be represented by different hardware modules (for example an *addition* could be implemented in hardware using a float adder or an integer adder), the visitor determines first, depending on the used data types and definitions from the *Triad* file, which module should be used, and delegates it to the appropriate handler.

If the CFG node is not a computation node, but describes branching or joining control flow, the base class takes care of it

Listing 4.11: Excerpt from the *Triad* generated visitor. The parts shown belong to the translation of an expression node, describing an addition.

```
    @Override
    public void visitAdditionExpr(AdditionExpr expr) {
        int handledBy = 0;
        if (alreadyTranslated.containsKey(expr))
            return;
        if (isHandledByHWOp_addfloat(expr)) {
            handledBy++;
            handleHWOp_addfloat(expr);
        }
        if (isHandledByHWOp_add(expr)) {
            handledBy++;
            handleHWOp_add(expr);
        }
        if (handledBy > 1) {
            System.err.println("Warning: Expr " + expr + " handled by more than
                 one visitor handlers");
        }
        if (handledBy == 0) {
            System.err.println("Warning: Expr " + expr + " not handled by a
                 visitor handler");
        }
    }

    private boolean isHandledByHWOp_add(AdditionExpr expr) {
        if (((expr.getCoreType().isIntegerType())
                && (( ((AtomicType)expr.getCoreType()).bitSize() == 1)
                    || ( ((AtomicType)expr.getCoreType()).bitSize() == 2)
                    [...]
                    || ( ((AtomicType)expr.getCoreType()).bitSize() == 64)))) {
            return true;
        }
        return false;
    }

    private boolean isHandledByHWOp_addfloat(AdditionExpr expr) {
        if (((expr.getCoreType().isFloatType())
                && (( ((AtomicType)expr.getCoreType()).bitSize() == 32)
                    || ( ((AtomicType)expr.getCoreType()).bitSize() == 64)))) {
            return true;
        }
        return false;
    }

    private void handleHWOp_add(AdditionExpr expr) {
        HardwareOperation_add op = new HardwareOperation_add();

        currentDP.subDatapath.addVertex(op);
        alreadyTranslated.put(expr, op.getSingleOutputPort());
       //set operator width
        op.setWidth(((AtomicType)expr.getCoreType()).bitSize());

       //visit the operands
        Expr [] operands = expr.getOperandArray();
       // treat input ports in the order they are defined in the triad config
        Expr opExpr = null;
        IOPort subOp = null;
        opExpr = expr.getOperand(0);
        opExpr.visit(this);
        subOp = alreadyTranslated.get(opExpr);
        op.connectInputPort(subOp, "A");
        addEdgeToSubDatapath(currentDP.getDatapath(), subOp.op, op);

        opExpr = expr.getOperand(1);
        opExpr.visit(this);
        subOp = alreadyTranslated.get(opExpr);
        op.connectInputPort(subOp, "B");
        addEdgeToSubDatapath(currentDP.getDatapath(), subOp.op, op);
    }
```

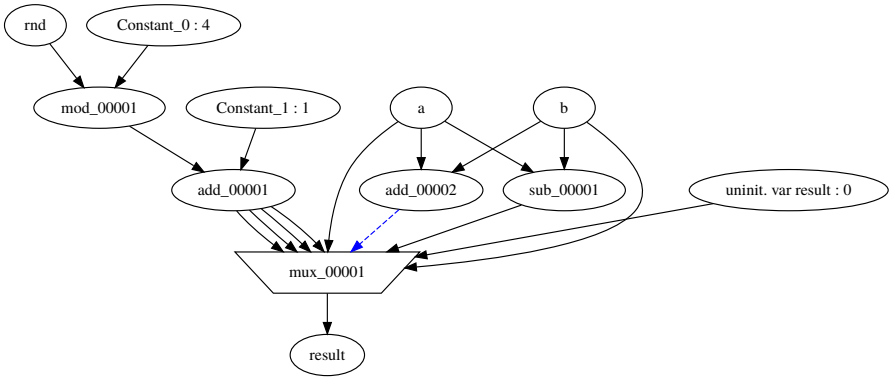


Figure 4.9: Internal CDFG representation after transforming the graph from Figure 4.8 into the IR of the generated backend.

and handles it (mostly these are ϕ expressions representing a join, which get translated into multiplexers). A map containing the already translated nodes is kept, to insert connectivity between the currently translated node and those previously translated nodes. In a few cases it can happen that nodes refer to yet untranslated nodes (for example this can happen in circular dependencies). Such cases are handled by inserting temporary dummy nodes. These dummy nodes get replaced in a later step by the actual translation of their nodes.

The complete transformation from CFG to CDFG of the program example from Listing 4.10 can be seen in Figures 4.8 and 4.9.

COMRADE 2.0 introduced memory edges to handle memory dependencies. These are modeled in COCOMA by inserting memory edges in program order at memory nodes (loads and stores). Thus, waiting for a token via a memory edge serializes the accesses (at most one at a time) and enforces the correct order of the memory accesses. A similar mechanism is implemented in *Triad*: The memory nodes are enumerated in program order. Predicates, which refer to the numbers, can be used in the rules and can so model these memory edges.

The next step that must be performed is the token controller generation from the rules. A token controller, shown as example in Figure 4.10, is located at each operator/hardware module; it starts the computation and signals the token flow. The token controller contains the storage for the tokens (encoded according to Section 4.2.4 with one-hot-encoding, binary-encoding, or a combination of both) called *Token Space Register,* (*TSR*). The number of places depends on the operation and the rules: for each input and for the outputs of the operation (i.e. most commonly three)

Listing 4.12: Pseudocode for the hardware generation in the backend.

```
Procedure Hardware Generation:
begin
  foreach CFG Subgraph c selected for hardware:
    TranslateScribble(c)
    SerializeMemoryAccesses()
    GenerateTokenController()
    OutputVerilogFiles()
end

Procedure TranslateScribble:
input: ScribbleCFG c
output: CDFG C

begin
  Identify loops L
  Sort L topologically (inside to outside)
  foreach loop l ∈ L:
    Build sub-CDFG s_l for l
       Visit(BeginNode(l))
    fix forward references
  Combine all s_l in to one big CDFG C
end


Procedure Visit:
input: Chord s (= Scribble node)

begin
  if s is a general computation
     Generate new hw ops node for CDFG according to hwops
         section
  else
     Generate node specific CDFG (phi node insert mux)
  Add s to alreadyTranslatedNodes(s, new CDFG node) map
  Visit(Children(n))
  ConnectToChildren(n)
end


Procedure SerializeMemoryAccesses:
begin
  Enumerate All Memory Mem accesses in program order
end
```

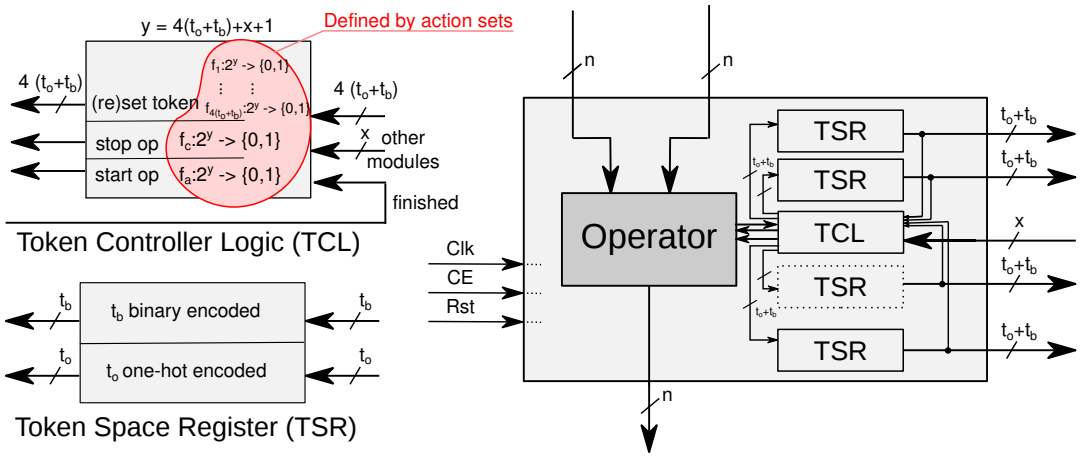


Figure 4.10: Generated wrapper around the hardware operators.

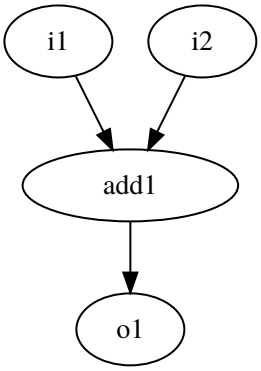


Figure 4.11: Very simple datapath.

a token place is allocated (when necessary). Another token place can be allocated for the node itself. This allows the node to have a state independent from its in- and outputs. Up to four actions can be triggered at each token controller by the *Token Control Logic* (*TCL*): creating a token, deleting a token, starting an operation and aborting an operation. All token signals are accessible from the outside. Thus, reading is a passive ability that requires no logic from the token controller. The other actions depend on some boolean functions, which combine different signals from different modules. In the process of the rules translation these terms are collected in sets, called *action sets*, which generate the input for the token memories and hardware module control signals.

This entire generation is the most complex step, and is shown as pseudo code in Listing 4.13, and illustrated with an example.

Figure 4.11 contains the most simple datapath possible, consisting of an add function (called `add1`), two global input opera-

tors (called `i1` and `i2`), and one global output operator (called `o1`). As notation, the action sets for a node n belonging to action a are called $A(n,a)$, and the (jth) input/output place are for a node n is called $p(n,I_j)/p(n,O_j)$.

To keep the example small, the focus is on the `add1` operator.

All three parts of a rule r, the entity selection E_r, the guard condition C_r and the actions A_r are used in this step. First, a set S containing the Cartesian product of all selected entities is computed. With a CFG as input, containing thousands of nodes, the resulting set size can become very large (Chapter 5 contains some statistics regarding this). Therefore, the entity subsets in E_r were introduced, reducing the size of the sets.

In the example, building the set S containing all elements of the Cartesian products for the first rule is quite easy. It is just one entity set, namely `Node`, that is selected in the *entity selection* part of the rule. So the elements in S are 1-tuple, and $S = \{$`add1, i1, i2, o1`$\}$.

Secondly, for each tuple in this set, the guard condition C_r is checked. The structural predicates (such as `isPred`) can be evaluated at compile-time, which means, that the variables from the tuple and from the quantifiers can be replaced by their boolean true/false value in the conditions. Should the rule evaluate at compile-time to a constantly *false* value, the rule does not apply to the node, and is ignored. If the condition is constant true, the actions from A_r are added to the relevant nodes action set. In case the condition is not constant, it has to contain a dynamic predicate, that must be evaluated at run-time (such as `hasToken`). The remaining logic term is added to the action set of the relevant node, and the process is repeated for all rules.

The results for the example can be seen in Table 4.6

As it can be seen clearly all the predicates in the guard condition do not evaluate to constant values, and so the complete term must be a run-time predicate, and therefore the term is added to the action set:

Listing 4.13: Pseudocode for the token controller generation from rules.

```
Procedure GenerateTokenController:
input: CDFG C, Rules R
begin
   foreach node n of C:
     foreach action a:
         action set s(n,a) ← ∅
   foreach rule r ∈ R:
     EntitySet S ← ∅
     // build Cartesian product
     foreach entity e ∈ entity selection term E_r from r:
         if (S == ∅)
           S ← e
         else
           S_old ← S
           S ← ∅
           for each t ∈ S_old:
             for each element f ∈ e:
               S ← S ∪ (t,f)
     // building rule for each entity tuple
     foreach entity tuple f ∈ S:
       Evaluate compile-time/generation-time predicates from
          guard condition C_r
       if (C_r not constant false)
          foreach  action A_r in rule:
             add to action set s(x,A_r) signals checking dynamic
                  condition

   foreach node n of C:
      foreach action a:
         check action set s(n,a) for conflicting rules
         check if action sets for create and delete are both
             empty
           or both non-empty
      // determine token space for n
      for each possible token place p
           determine number of token types in
                    delete/create action set for n, p
      cache existing controller types
      output token controller
end
```

$e \in S$	unevaluated guard condition(e)	compile-time evaluated guard condition(e)
`add1`	hasToken(p(`add1`,I_1),*Activate*) $\wedge$ hasToken(p(`add1`,I_2),*Activate*)	hasToken(p(`add1`,I_1),*Activate*) $\wedge$ hasToken(p(`add1`,I_2),*Activate*)
`i1`	...	...
`i2`	...	...
`o1`	...	...

Table 4.6: Elements selected by entity selection, the guard condition applied to them, and evaluated at compile-time. No guard condition evaluates at compile-time to a constant value.

$$
\begin{aligned}
A(\texttt{add1}, start) = &\{hasToken(p(\texttt{add1}, I_1), Activate) \\
&\wedge hasToken(p(\texttt{add1}, I_2), Activate)\} \\
A(\texttt{add1}, delete(p(\texttt{add1}, I_1), Activate)) = & \\
&\{hasToken(p(\texttt{add1}, I_1), Activate) \\
&\wedge hasToken(p(\texttt{add1}, I_2), Activate)\} \\
A(\texttt{add1}, delete(p(\texttt{add1}, I_2), Activate)) = & \\
&\{hasToken(p(\texttt{add1}, I_1), Activate) \\
&\wedge hasToken(p(\texttt{add1}, I_2), Activate)\}
\end{aligned}
$$

As already mentioned, the sets for `i1,i2` and `o1` are omitted for clarity. Now the second rule is handled, therefore the set containing the Cartesian product of the sets selected with the *entity selection* part of the rule is built: First, the set S is initialized to the empty set, and gets assigned all elements of the `Node` entity set in the first pass, similar to the processing of the first rule, resulting in $S = \{$`add1, i1, i2, o1`$\}$.

As there is another entity set E' in the *entity selection* this time, it is $S_{old} = S$ assigned, and $S = \emptyset$ again. Now for each element $s \in S_{old}$ all tuples (s, e) $e \in E'$ are entered into S, generating the Cartesian product $S \times E'$ resulting in the set shown in Table 4.7.

As can be clearly seen the set S grows in the power of the number of elements of the entity selection. Keeping it small reduces the memory footprint and the runtime of the compiler.

The application of the condition gives Table 4.8.

The resulting term is added to the action set:

$$A(\texttt{add1}, create(p(\texttt{add1}, O_1), Activate)) = \{isFinished(add1)\}$$

$$
\begin{aligned}
S = \{ \quad & (\texttt{add1}, p(\texttt{add1}, O_1)), \quad (\texttt{add1}, p(\texttt{i1}, O_1)), \quad (\texttt{add1}, p(\texttt{i2}, O_1)), \\
& (\texttt{add1}, p(\texttt{o1}, O_1)), \quad (\texttt{i1}, p(\texttt{add1}, O_1)), \quad (\texttt{i1}, p(\texttt{i1}, O_1)), \\
& (\texttt{i1}, p(\texttt{i2}, O_1)), \quad (\texttt{i1}, p(\texttt{o1}, O_1)), \quad (\texttt{i2}, p(\texttt{add1}, O_1)), \\
& (\texttt{i2}, p(\texttt{i1}, O_1)), \quad (\texttt{i2}, p(\texttt{i2}, O_1)), \quad (\texttt{i2}, p(\texttt{o1}, O_1)), \\
& (\texttt{o1}, p(\texttt{add1}, O_1)), \quad (\texttt{o1}, p(\texttt{i1}, O_1)), \quad (\texttt{o1}, p(\texttt{i2}, O_1)), \\
& (\texttt{o1}, p(\texttt{o1}, O_1)), \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}
$$

Table 4.7: Generation of the Cartesian Product of the by *entity selection* selected sets.

$e \in S$	guard condition(e)	compile-time evaluated condition(e)
$(\texttt{add1}, p(\texttt{add1}, O_1))$	isOutputOf($p(\texttt{add1}, O_1)$) $\wedge$ isFinished(`add1`)	isFinished(`add1`)
$(\texttt{add1}, p(\texttt{i1}, O_1))$	isOutputOf($p(\texttt{i1}, O_1)$) $\wedge$ isFinished(`add1`)	*false*
$(\texttt{add1}, p(\texttt{i2}, O_1))$	isOutputOf($p(\texttt{i2}, O_1)$) $\wedge$ isFinished(`add1`)	*false*
$(\texttt{add1}, p(\texttt{o1}, O_1))$	isOutputOf($p(\texttt{o1}, O_1)$) $\wedge$ isFinished(`add1`)	*false*
`(i1, …)`	…	…
`(i2, …)`	…	…
`(o1, …)`	…	…

Table 4.8: Elements selected by entity selection, the guard condition applied to them, and evaluated at compile-time. Some of them evaluate at compile-time to a constant *false* and have no further impact.

Except one all, of these evaluations (when just observing the `add1` node) are constantly *false* at compile-time, and can be discarded. At the end, one additional term is added to an action set of `add1`. For the sake of brevity, the last rule is skipped.

Afterwards, a compile-time check is performed on the generated action sets: It is checked if creation and deletion of tokens match (i.e. must be different conditions deleting and creating a token of the same type). When the check is passed, the token space is determined (that is, it is checked which tokens are actually used at the action sets of a node). Each exclusive token set gets its own register, so for each set of mutually exclusive tokens, only one can be present, but tokens in different sets can be present at the same time.

This example is small, each of the resulting action sets has just one term, and the create and delete action are not triggered on the same terms. The terms of creation and deletion can, even when different, still both evaluate at run-time to *true*, but this is not detected at compile-time.

Additionally a check is performed if create and delete action are both non-empty (or both empty). Should only one of the two be empty, an error is thrown. In this example this would happen, as it is incomplete: the third rule was not evaluated, and the output place has nothing to trigger a delete action, and the input place has no create action. With the third rule this problem is solved, and the token spaces would be determined: In this case, the output and the inputs have a place for one token, which would result in three token places with one bit of storage each.

In the hardware generation process the terms of an action set get *or*'-ed together and become the input to the token memories/hardware module (the combination of *create* and *delete* signals is done with $value_{new} = (create \vee value_{old}) \wedge \overline{delete}$). When the binary encoding is used, additional logic is necessary: required en- and decoders are added.

Once the logic for a token controller is built, it is checked if the same token controller has already been built before, and if not, it is stored with a new identifier (actually just an increasing number) for its output. If it already existed, the newly created is discarded, and just the identifier is used. Two token controllers are considered to be the same if they have the same logic (i.e. the token sets have the same boolean terms); the connectivity of the logic (i.e. to which signals the boolean term inputs are connected to) is not relevant.

Also each node gets a custom *start*/*finish* logic when the node is not a variable latency operator. As other operators start one operation every cycle, no start and finishing signaling required, and the compiler uses a shift register of a length of the given latency of the operator to keep track of input data.

Finally, the generated token controllers are emitted as Verilog code into one module, which get selected via a type parameter instantiated by the hardware wrapper module of the operators. Figure 4.10 shows the block diagram of such a module with its token controller.

5 EVALUATION

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

WINSTON CHURCHILL,
Politician

Section 4.2.9 already gave a small example rule set for a simple token flow model and as already described there, one token is not enough to handle control flow, but far more and complex rules are required. Originally motivated by the *COCOMA* token flow, the implementation of rules resembling those micro architecture, was the main focus. So as first part of the evaluation the implementation of *COCOMA*-style controllers is presented and tested with several examples from a benchmark suite.

The aim of the complete environment around *Triad* was not to have a C-to-HDL compiler that can compete with others, but to have a compiler, that allows easy exploration of different compiler variants. Of course it is hard to come up with a metric for *easy*. To perform some exploration some variants are generated, the necessary changes in the *Triad* file is given. The resulting compilers are tested with the same examples and compared to the result of the base compiler.

A detailed evaluation of the run-time behavior of the compilers was omitted. The execution time of both compilers (*Triad* and *hardScale*) were negligible. *Triad* needs less than 10 seconds to generate a new compiler. More impact has the used java compiler, as the generated backend must be compiled into a new *hardScale*. But even this takes less than a minute. The execution time of *hardScale* is for bigger examples tenth of seconds, still an insignificant amount considering the time the synthesis requires.

5.1 IMPLEMENTATION OF *COCOMA*-BASED RULE SET

COCOMA used two types of tokens, *activate* tokens (AT) to activate operations, and *cancel* tokens (CT) to remove *activate* to-

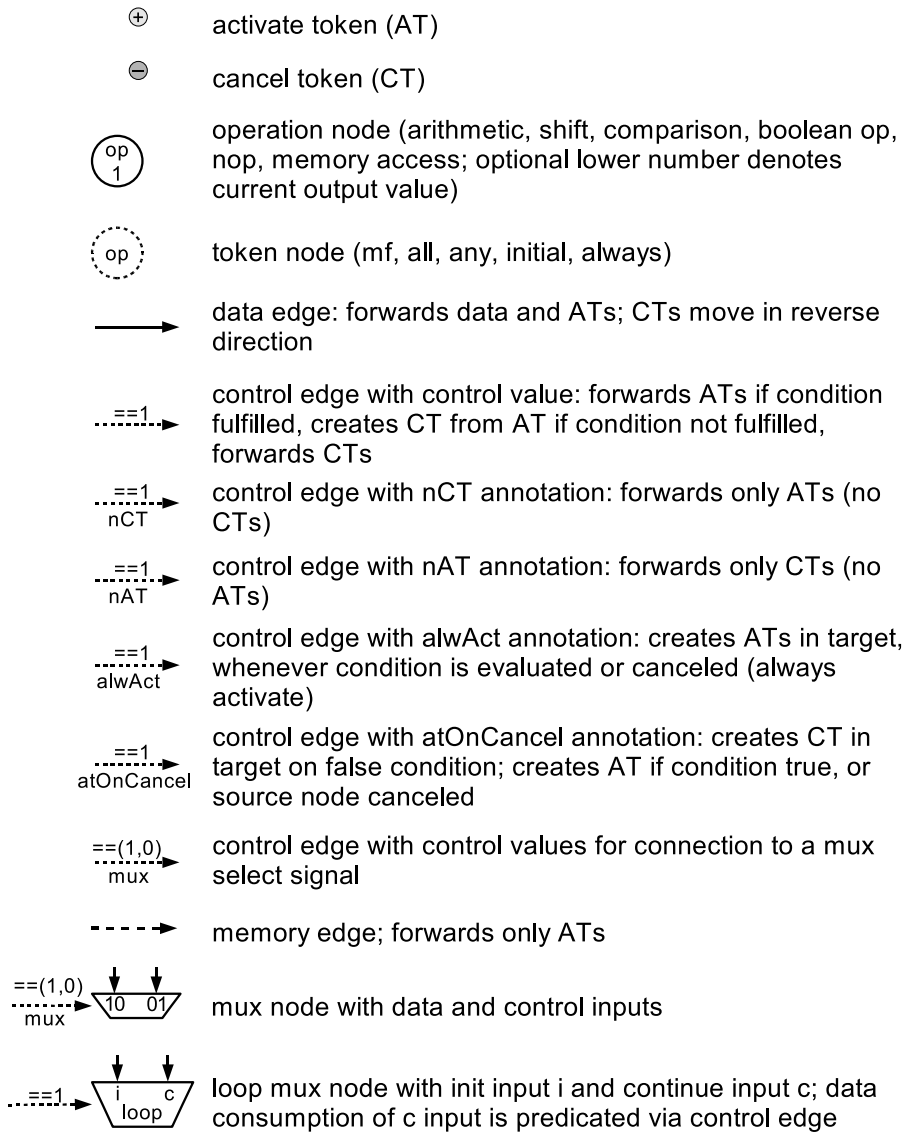


Figure 5.1: Legend for graphical *COCOMA* token flow rules in Figure 5.2 (from [67])

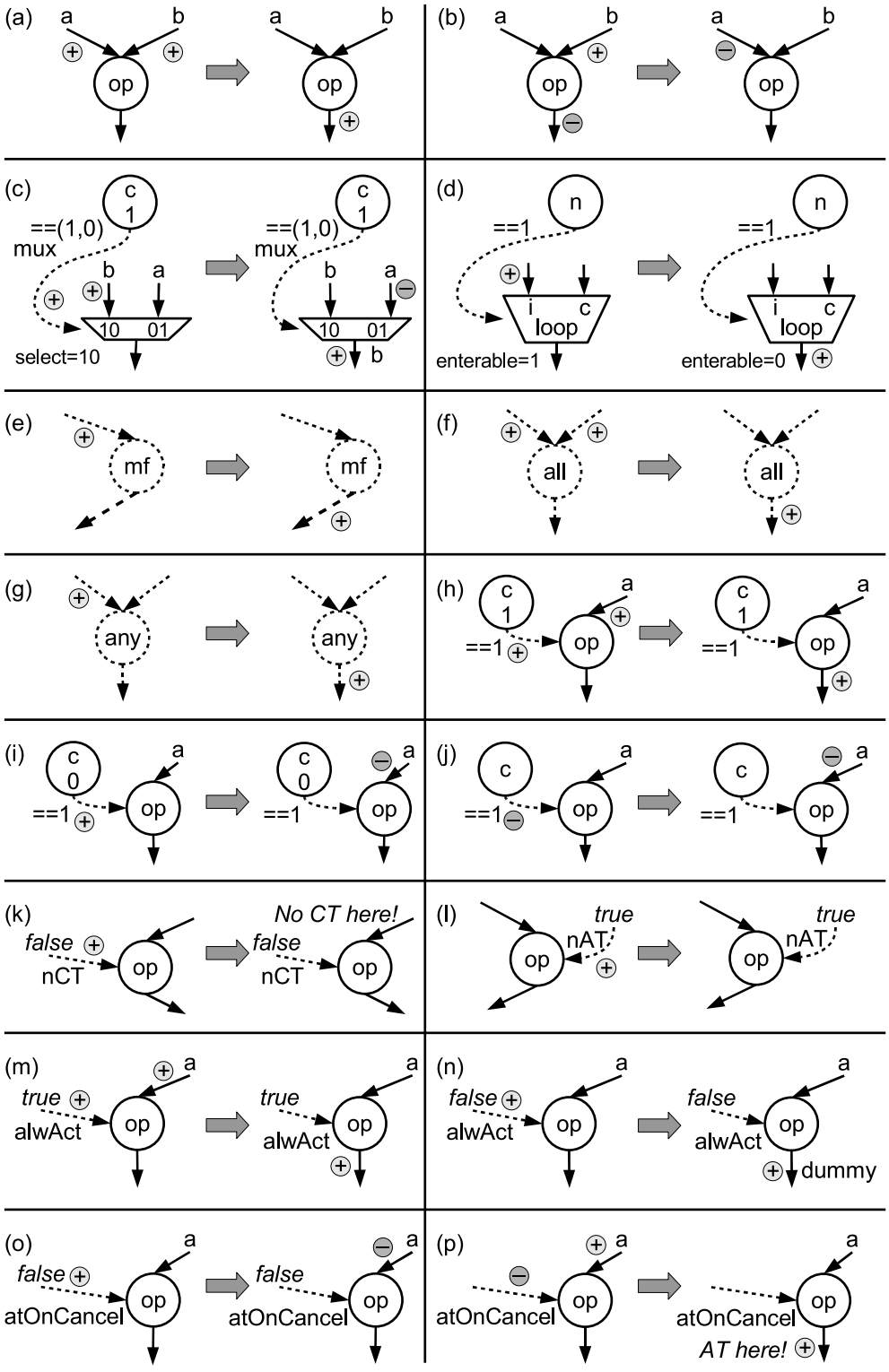


Figure 5.2: Graphical *COCOMA* token flow rules (from [67])

kens from misspeculated branches. These *CT* flow in opposite direction of the *AT* and when they meet they cancel each other out. To avoid dead locks, handling of all special cases, and correct serialization of memory dependencies, several extra annotations to the CMDFG were introduced, namely: *nCT* (no cancel token), *nAT* (no activate token), *alwAct* (always activate), *atOnCancel* (creates activate when source node is canceled). Further, several additional node types were introduced to handle flow of tokens along the control edges (*mf* memory forward node, *all* a node that behaves like a boolean *and* for tokens, *any* a node that behaves like a boolean *or* for tokens, and and *always* node that generates every cycle an *AT*). These were necessary as the control edges were used also for purposes different than control flow (like the memory serialization).

Figures 5.1 and 5.2 are taken from [67] and give an overview of the token flow in *COCOMA*. The different rules describe the behavior of the tokens at different nodes or under different conditions. Explaining all of the rules is out of scope of this thesis. To illustrate the behavior of the controller, some of the more important rules are described.

For example, Figure 5.2(a) describes that an operation starts, when both inputs have an *AT*. When doing so, the *AT* at the inputs get removed and one *AT* is created at the output. Figure 5.2(b) describes that when an operator receives at its output a *CT* it cancels an existing *AT* at the input. The *CT* gets forwarded to the inputs were no *AT* was waiting. A more complex case is shown in Figure 5.2(c). It is hardware generated from an SSA ϕ node, where node *c* contains the condition, that decides which of the parameters (*a* or *b*) gets selected. An *AT* along the control edge decides which of the inputs is taken, and when the corresponding input gets an *AT* it gets forwarded to the output and at the not selected input is an *CT* injected.

The implementation of the rules can be seen in Appendix D, with comments indicating which *COCOMA*-rule they refer to, when possible. One remarkable point is that the *Triad* rule set uses more than two tokens. One of *COCOMA*s node type can have an internal state, which is easily modeled by an additional token, just at this node. The other token is introduced, because *COCOMA* uses the control edges for almost anything, except data flow. The result is that different aspects interfere. To reduce the problems (and making the rules a little bit easier) a token for the memory serialization was added, instead to model it with *AT* along control edges (which therefore need special annotations

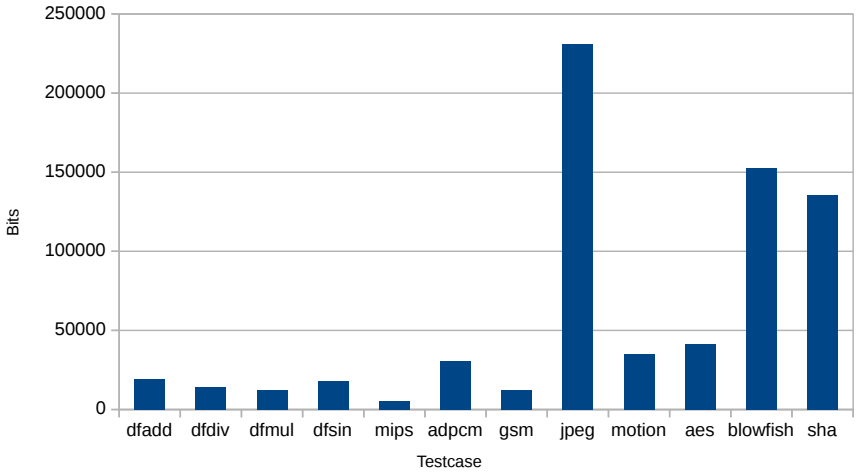


Figure 5.3: Memory requirements for the testcases.

to handle special cases). A further iteration of the rules could remove even more of this behavior.

5.2 TESTCASES AND ENVIRONMENT

For testing purposes a benchmark suite was utilized. *CHStone*[85, 86] is a HLS-suite, and is specifically aimed to test HLS compiler. The function that is supposed to be implemented in hardware has to be manually identified. It was chosen so that it included the topmost function excluding the test frame added from *CHStone*.

Table 5.1 gives an overview of the testcases in the benchmark suite used together with their source code characteristics. It shows that the testcases in the benchmark suite try to cover many different programs: Some have many branches compared to the computations, some are almost sequential, some have few loops, and some many.

The hardware operator library was a stripped down version of the *modlib* library from Section 3.3.3 utilized for these tests. Stripped down means, in this case, that the token handling logic, originally added for *COMRADE 2.0* to the library, was removed again. The *hardScale* compiler will automatically generate its own token handling logic to the operators.

The memory footprint of the benchmarks is small enough (see Figure 5.3), that a simple local memory instantiated in Block RAMS on the FPGA device was sufficient to hold all data. This

Example	Description	Functions	Source Code Operations					Statements		
			add/sub	mul	div	compare	logic	loops	branches	assign.
dfadd	double addition	4	38			78	146	1	87	299
dfdiv	double division	4	45	8	2	50	73	3	47	220
dfmul	double multiplication	4	28	4		34	61	1	38	159
dfsin	double sine function	3	141	17	2	196	357	4	216	864
mips	simplified MIPS processor	5	17	2		12	23	4	6	66
adpcm	adaptive differential PCM	26	156	69	2	73	24	25	76	792
gsm	linear pred. coding analysis	12	251	53		110	41	18	95	492
jpeg	image decompression	30	1029	148	6	242	132	117	277	266
motion	motion vector decoding	13	299			155	55	71	115	1100
aes	adv. encryption standard	11	510	22	36	48	370	24	36	909
blowfish	data encryption standard	6	280			15	370	10	13	385
sha	secure hash algorithm	8	134		3	32	87	29	2	315

Table 5.1: Software characteristics of the CHStone benchmark example [86]

Testcase	LUTs	FF	DSPs	BRAMs	Clock	Cycles
adpcm	24755	21380	145	12	150	50812
aes	7472	4312	0	11	125	10784
blowfish	4560	2533	0	27	125	453756
dfadd	3495	3037	0	0	175	1694
dfdiv	7199	5312	61	2	150	3014
dfmul	2311	1127	17	0	125	538
dfsin	20484	12270	90	5	125	59556
gsm	8468	6687	56	14	125	13916
jpeg	33429	17466	15	81	125	2667620
mips	2638	1021	8	5	125	17602
motion	2387	1853	0	4	175	15098
sha	5328	3760	0	15	150	389894

Table 5.2: Synthesis results for the CHStone testcases.

common practice for HLS benchmarking removes any influence of the memory system and focuses on the generated datapath.

For functional verification of the generated hardware it was synthesized and placed & routed with Xilinx *Vivado 17.4* on a Virtex 7 VX690T FPGA. As *Vivado* tries to reach the target frequency and not the highest possible, the synthesis runs were repeated with different target frequencies with a stepping of 25 MHz, until the highest possible frequency $f_{\max}$ was found. To measure the run-time, the testbench counts the cycles from issuing the start signal, until the hardware signals the end of the computation. Product of both, $f_{\max}$ and cycles, would be the total execution time.

5.3 SYNTHESIS RESULTS

Table 5.2 gives the result of the synthesis for the rules set from Appendix D.1. The numbers are not very good; for example, a comparison with *Nymble-ST* from [95] shows that *hardScale* hardware is on average 4.29 times slower than *Nymble-ST*s (for the *CHStone* testcases).

On one hand the memory system is responsible for this. While *hardScale* already uses a very simple system, *Nymble-ST* uses an even more simplified, non-synthesizable, memory system. This

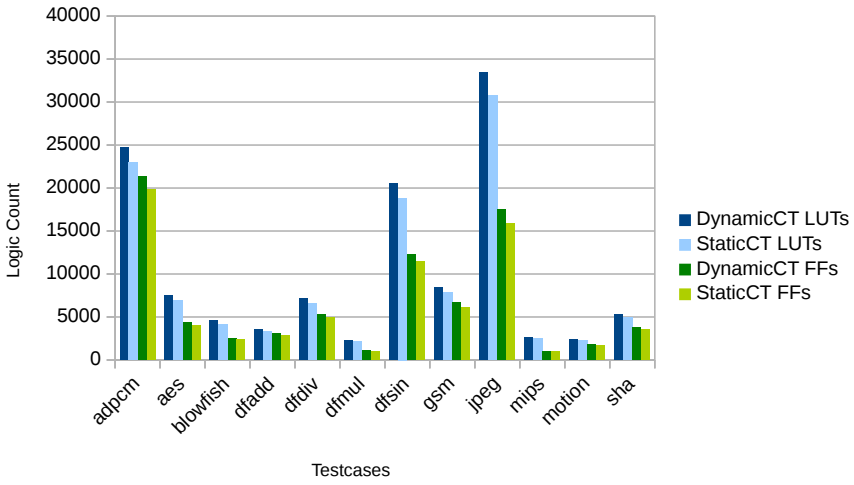


Figure 5.4: Standard *COCOMA* token flow with dynamic *cancel* tokens compared to static *cancel* token waiting at the multiplexer. Run-time did not change.

not only allows an access to complete in one cycle, but also an infinite number of parallel accesses.

On the other hand the token based approach suffers in its current implementation from the *bubble*[1] problem: After an operator has finished its computation, the creation and forwarding of the token requires one additional cycle, in which the next operator is idling.

5.4 TOKEN MODEL VARIANTS

One advantage of the approach proposed in this thesis, is that different variants can be easily explored. After having shown a very simple data flow model (that does not allow memory accesses in branches) in Section 4.2.9 and the regular *COCOMA* implementation discussed in 5.1, one further variants is easily implemented using *Triad*.

In a conventionally developed HLS system, many weeks would be required from reasoning about the new control scheme to having a functional HLS system implementing that scheme.

With some changes of the rules (see Appendix D.2), the token flow behavior is changed. For the *Pegasus* compiler [26] it is the regular behavior, and *COMRADE 2.0* used it later also as optional variant. [67] introduced an alternative to the standard

1 The term comes from token models where the token are considered to flow down, while the empty space float upwards, similar to a bubble in the water

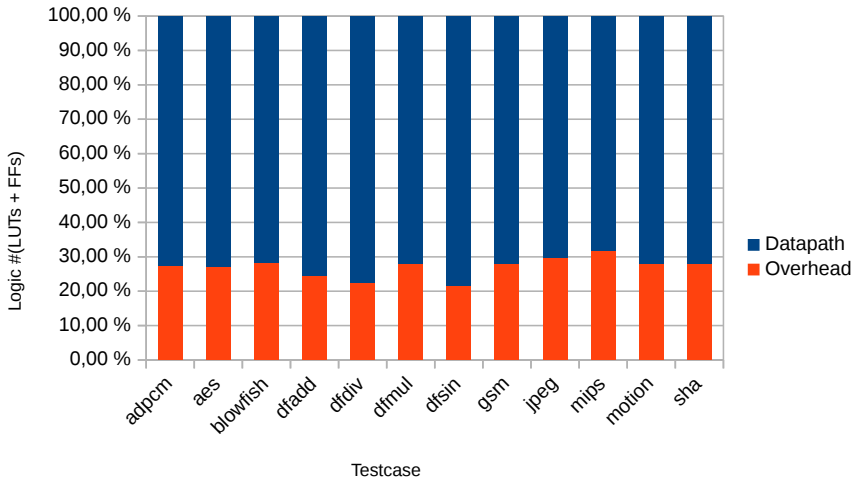


Figure 5.5: Comparison of required resources from data path with and without token logic.

COCOMA token flow. Here, the *CT* do not flow dynamically in opposite direction of the *AT*, but instead wait statically at the multiplexer inputs.

The results, visualized in Figure 5.4, show very clear that the static token model is better. While having no impact on the execution time, it requires on average 8% less LUTs and 11% less FFs. These numbers back up the results from Gädke, who also came to the conclusion that the static approach is superior to the dynamic.

5.5 OPTIMIZATION

One main disadvantage with the tokens, is the additional logic that is required at each operator. A synthesis run of the data path, just with the operators, but without any token logic, shows an average overhead of 27 % (see Figure 5.5) for the token logic.

One way to reduce the overhead is to make the token logic not only responsible for one operator, but instead to multiple operators. In the CDFG from which the hardware is build, subgraphs can be created that consist only of data flow with operators that have constant latency. All of the operators inside can be *statically* scheduled, and are treated as a single *virtual* operator. This region of the graph now gets only one instance of the token logic and the overall overhead is reduced.

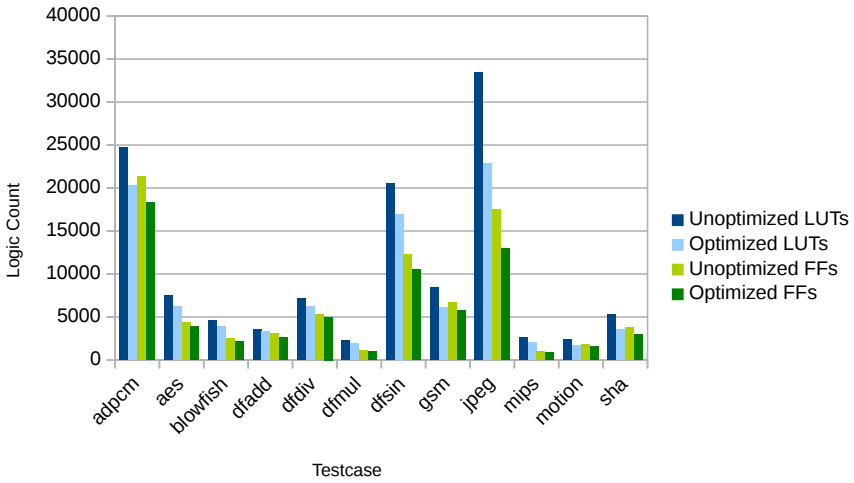


Figure 5.6: Reducing the token logic induced logic overhead by treating subgraphs as virtual operators.

Finding the subgraphs is simple. A greedy algorithm, traversing the graph finds all of them. Figure 5.6 shows an example of such an optimization.

The result of the optimization is shown in Figure 5.6. Compared to the CDFG without the optimization, the new version requires (on average) 21 % less LUTs and 14 % less FFs, while the number of DSPs and BRAMs did not change.

6 SUMMARY AND FUTURE WORK

I didn't have time to write a short letter, so I write a long one instead.

MARK TWAIN, Writer

In this work, in Section 3.2 and Appendix B, a broad survey of all existing C-to-HDL compiler was presented together with an introduction to the basic concepts of those (in Chapter 2). Due to the sheer number of compilers (quantity over quality) only *COMRADE* (and its predecessors) was discussed in greater detail in Section 3.3. *COMRADE* was chosen as it is one of the few compilers that generates hardware which is dynamically scheduled.

Suffering from poor comparability, as each compiler is different in almost every aspect (that is in their underlying generated micro architecture, their scheduling method, their applied optimizations, their execution model, their target hardware, and their input language restrictions and extensions) and from the problems experienced when working with *COMRADE*, the implementation of *Triad* was proposed in Chapter 4. It should allow for easy exploration and comparison of token based dynamically scheduled datapaths.

Triad generates a hardware generation backend for the *Scale* compiler framework. The generated backend uses a scheduling method that is inferred from a mathematical set of rules parsed by *Triad*. With the description of the implementation in Sections 4.2 and 4.3 a small example for the rule set was presented.

In the evaluation in Chapter 5 the results for a set of standard benchmarks were given. While not particularly fast, the generated hardware was more than four times slower compared to hardware from other compilers, it showed its flexibility. After implementing a complex token based scheduling method for the first test, it was very easy to explore different a variant. With small changes in the rules, it was easily possible to compare both versions and identifying the variant that requires up to 11 % less hardware resources.

In Section 5.5 an optimization to the generated datapath was presented. It reduces the required hardware resources by up to 21 % by reducing the number of required control logic.

Possible future work can be divided into two different aspects: One deals with possible improvements that increase the flexibility of the *Triad* even more, by allowing for more variation in the generated hardware. Especially interesting would be support of a (or more than one) pure static scheduling mechanism.

As the generated hardware is not as fast as others, the other aspect covers the speed of the generated hardware. *Triad* could include further optimizations, which already exist in other hardware generation compilers. For example, *COMRADE 2.0* contains queues at the in- and outputs, which reduce the back pressure and allow faster execution at the expense of more hardware resources. Software service calls are another feature not yet supported (and one which *COMRADE 1.0* already had).

One way to improve the datapath is to use operator chaining: Small operators in sequence can be connected combinational without registers between them. Lenient execution, that is starting an operation when the input is only partially available (like one boolean *false* value as adder input), could also improve the speed of the generated hardware.

Another part which could be improved, is the usage of the memory backend. While different memory backends can be used, as long as they support a read and write module, the generated compiler always serializes the accesses and allows only a single memory access at the same time (while the used BRAM based local memory is dual ported and would already support two at the same time).

Finally, the whole system could be improved when more advanced checks of rules would be implemented (for example trying to proof that the token flow is dead lock free).

Concluding, it can be summarized that the *Triad* compiler can be used to generate hardware compilers from a formal rule set. While the resulting hardware is not the fastest, the *Triad* environment is far more flexible than traditional C-to-HDL compilers and allows for a higher abstraction level when dealing with token based micro architectures for scheduling.

A

FORMAL DEFINITIONS

Definition 1 (Basic Blocks)
A Basic Block (BB) is a sequence of program statements without branches within the statements – at most one branch as last statement in the block is allowed. The sequence of program statements may be empty.

Definition 2 (Control Flow Frame)
A Control Flow Frame (CFF) is a 5-tuple $(N, E, cond, start, end)$, with

- *N, E two finite sets, describing a directed connected graph $G = (N, E)$ where N is the node set and $E \subseteq N \times N$ the edge relation,*
- *a condition function $cond : E \to \boldsymbol{Z} \cup \{\epsilon\}$,*
- *a start node $start \in N$, with the property that every node $n \in N$ is reachable from start, and*
- *an end node $end \in N$, with the property that end is reachable from every node $n \in N$.*

Definition 3 (Control Flow Graph)
A control flow graph C is a triple $C = (CFF, BB, map)$, with

- *CFF being a CFF,*
- *BB being a set of basic blocks, and*
- *$map : N \to BB$ a mapping function, that maps CFF nodes to BBs.*

Definition 4 (Branch Node)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: A node $n \in N$ is a branch node, when its out degree > 1.

Definition 5 (Join Node)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: A node $n \in N$ is a join node, when its in degree > 1.

Definition 6 (Region)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: A subset of nodes $R \subset N$ is called a region.

Definition 7 (Dominates Relation)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: A node $n_1 \in N$ dominates, or is called dominator of node $n_2 \in N$, if every path from start to n_2 contains n_1.

Definition 8 (Post-dominates Relation)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: A node $n_1 \in N$ post-dominates, or is called post-dominator of node $n_2 \in N$ if every path from n_2 to end contains n_1.

Definition 9 (Dominance Frontier)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: The Dominance Frontier (DF) of a node n is defined as the set of Nodes $DF(n) \subset N$, with the property $\forall f \in DF(n) : f$ is not dominator of n, but f has a successor node $g \in N$ that dominates n.

Definition 10 (Post-Dominance Frontier:)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: The Post-Dominance Frontier (PDF) of a node n is defined as the set of Nodes $PDF(n) \subset N$, with the property $\forall f \in PDF(n) : f$ is not post-dominator of n, but f has a predecessor node $g \in N$ that post-dominates n.

Definition 11 (Control Dependence)
Let $C = ((N, E, cond, start, end), BB, map)$ be a CFG, then we define: Node $c \in N$ is the controller of (controls) Node $n \in N$ if $c \in PDF(n)$. n is also called control dependent on c.

Definition 12 (t-structured)
Let $C = ((N, E, ann, start, end), BB, map)$ be a CFG, then C is called t(op)-structured*, when all loop conditions are evaluated at the top. That is, their CFF loop subgraph is isomorph to the graph in Figure 2.6.*

Definition 13 (Back-edge)
Let $C = ((N, E, ann, start, end), BB, map)$ be a t-structured CFG, then we define: An edge $e = (n_1, n_2) \in E$ is called back-edge when n_2 dominates n_1.

Definition 14 (Loop Header)
Let $C = ((N, E, ann, start, end), BB, map)$ be a t-structured CFG, then we define: A node $l \in N$ is called loop header, when l has an incoming back-edge.

Definition 15 (Loop Body)
Let $C = ((N, E, ann, start, end), BB, map)$ be a t-structured CFG, then we define: When $l \in N$ is a loop header and $e = (n, l)$ is its back edge, then the Loop Body $(LB)(l)$ of l are all nodes on all path between l and n.

Definition 16 (SSA-Form)
Statements are in static single assignment form, when each variable has exact one static definition (that is, it is written only once), and each usage refers to exactly one variable.

Definition 17 (Data Flow Graph)
A Data Flow Graph (DFG) is a 5-tuple (N, E, V, OP, map) where

N: A set of nodes.

E: A set of edges $\subseteq N \times N$.

V: Set with possible value of the data items.

OP: A set of functions $op \in OP, op : V^i \rightarrow V^o$, with $i = indegree(map(op))$ and $o = outdegree(map(op))$.

map: $map : N \rightarrow OP$ a function mapping the nodes to the operations.

(N, E) is a directed acyclic graph. At each node n the operation $o = map(n)$ is applied to the data values from the predecessor nodes at the incoming edges and the result goes via the outgoing edges to the successor nodes.

B

LIST OF C-TO-HDL-COMPILERS

B.1 XPILOT

OTHER NAMES / RELATED COMPILERS: AutoPilot, AutoESL, Vivado HLS

CLASSIFICATION: academic, commercial, static, manual, hwsw, control flow

AUTHORS / COMPANY: Originally University of California, Los Angeles as xPilot. Later as AutoESL, from AutoESL Design Technologies. This got bought 2011 by Xilinx, which incorporated AutoESL into their Vivado Tool Suite. 2006

REFERENCES: [44, 45]

TARGET: Conventional FPGAs (later Xilinx only)

DESCRIPTION: Started as academic xPilot. Besides C it accepts also C++ and SystemC as input language, and generates Verilog HDL or VHDL from it. The scheduling is done by formulating the scheduling as constraints for Linear Program (LP), which get solved by a dedicated LP solver. These are generalized ASAP and ALAP schedules and are optimized for worst-case longest-path latency, expected average-case latency and the overall slack distribution.

Nested conditions and loops work and control flow is implemented by using a state machine. In the early xPilot the C operators were just translated into Verilog HDL operators, which results in unsynthesizable code in case of the division operator (`/`). In newer versions this limitation is gone, even floating point and different bit widths are support and operators can be shared, too.

Common optimizations, like loop unrolling and dead code elimination, are applied as well as pipelining. Also an overall optimization to the target system happens, by using area and delay information for selecting the implementing operator logic and apply these information to the scheduling constraints.

Uses the LLVM [121] framework as IR.

RESTRICTIONS: Pointer support is only very limited.

B.2 BACH-C

CLASSIFICATION: commercial, static

AUTHORS/COMPANY: Sharp 1997

REFERENCES: [161, 178]

TARGET: ASIC

DESCRIPTION: Additional commands (e.g. `par` and `chan` to model parallel execution and synchronous communication) are present and are necessary to translate a program. The semantics for parallelism and communication are based on Communicating Sequential Processes (CSP) [90]. This is also more a modification of the C language (called Bach-C) to perform hardware description instead of a way to translate legacy code.

RESTRICTIONS: Pointers are not supported (but array accesses are).

B.3 C2H

CLASSIFICATION: commercial, static, manual, HWSW

AUTHORS/COMPANY: Altera 2005

REFERENCES: [122], [6]

TARGET: NIOS II soft-core cpu for Altera FPGAs

DESCRIPTION: The compiler can translate (manually selected) C-code at function level into hardware accelerators. The generated hardware are not dedicated IP blocks, but custom peripherals which become part of the ALU of the soft-core processor. Therefore, they are limited to functions with two inputs and one output. So these hardware accelerators are more a kind of custom instruction set enhancement. In contrast to many other compilers, pointer

dereferencing is possible, via Avalon-MM master ports that share the memory connection with the soft core-CPU. Abandoned 2013 and replaced by Altera SDK for OpenCL.

RESTRICTIONS: In addition to the two inputs, one output restriction many other things are not supported in the C code: Floating point, recursion, `goto`, address operator `&`, and short circuit evaluation.

B.4 ALTERA SDK FOR OPENCL

OTHER NAMES / RELATED COMPILERS: AOCL, Intel HLS Compiler

CLASSIFICATION: commercial, manual, HWSW, static

AUTHORS / COMPANY: Altera/Intel 2012 (Intel 2016)

REFERENCES: [7, 54, 102, 103]

TARGET: Altera/Intel FPGA accelerator boards

DESCRIPTION: In a preconfigured board environment a translated OpenCL kernel gets loaded and executed. The recommended usage of this compiler is to use just one work-item and gain parallel execution speed by deep data path pipelines. Uses the LLVM compiler infrastructure [121] to generate an optimized LLVM IR, which is used to generate a CDFG. Using this CDFG representation Verilog HDL code for the data path is created.

RESTRICTIONS: Accepts not generic C, but OpenCL 1.0 as input, including the generic OpenCL restrictions: No function pointer, no recursion, no variable length arrays. The "compiling" of the kernel can take many hours, as this includes the synthesis and place and route of the hardware accelerator. While this is expected behavior for a hardware compiler, it is unusual for an OpenCL compiler.

B.5 C2R

CLASSIFICATION: commercial, static, hwsw, manual

AUTHORS / COMPANY: CebaTech, renamed to Altior in 2011. In 2013 Altior was aquired by Exar Corp. 2004

REFERENCES: [2, 30]

TARGET: Generates Verilog HDL code, useable for FPGA and ASIC.

DESCRIPTION: Not much information is available, as Exar does no longer offer the compiler. It extends C with directives that moves C closer to a HDL, like instructions for interfacing and for pointer/memory association. With the known restrictions, it behaves less as an automatic hardware generation compiler, but more like a high level, C based, hardware description language that has the necessary libraries that it can also be compiled as C program.

RESTRICTIONS: The existing documentation shows that C code has to be restructured and the interfacing with the hardware is not generated automatically, but library calls for invocation must be manually inserted. Also memory has to be tagged, to describe how it is handled, or where it should be placed. Supports pointer and floating point.

B.6 C2VERILOG

OTHER NAMES / RELATED COMPILERS: C Level Design

CLASSIFICATION: commercial, static, HW

AUTHORS / COMPANY: CompiLogic renamed to *C Level Design* (1998), ceased operation 2001 (Synopsys bought its assets, but continued only simulation technology) 1996

REFERENCES: [146]

TARGET: FPGA and ASIC.

DESCRIPTION: As the original company is no longer existing and Synopsys discontinued the HLS compiler, not much information is available. From old documents and marketing articles it seems, that they supported almost all C constructs, including pointers and loops. It was also able to generated test benches for the hardware.

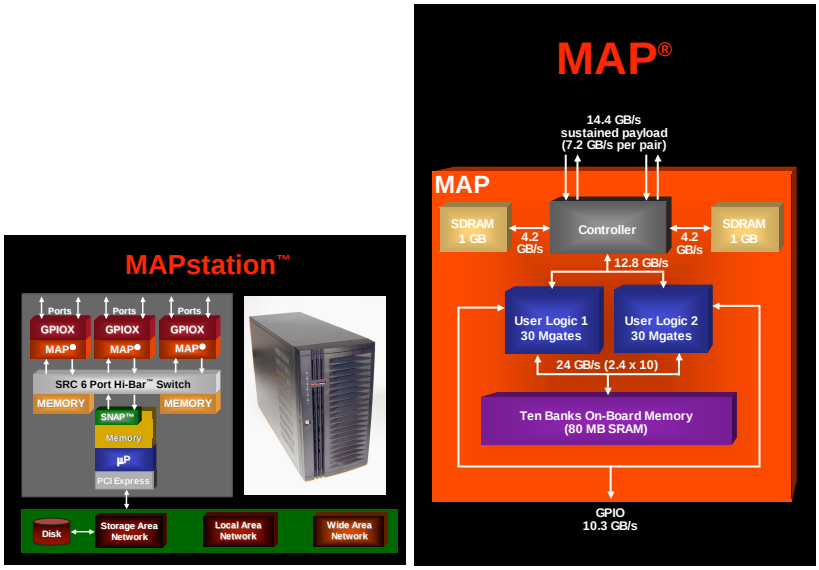


Figure B.1: Diagram of a SRC Computers SRC-7 MAPstation and a single MAP element (from [92]).

RESTRICTIONS: Users reported that `printf` are translated into `write` and that certain operators have very long combinational paths, which led to bad frequencies.

B.7 CARTE

OTHER NAMES / RELATED COMPILERS: Carte++, (uses Altera FP Compiler)

CLASSIFICATION: commercial, hwsw, automatic

AUTHORS / COMPANY: SRC Computers LLC 1996

REFERENCES: [112, 113]

TARGET: Uses SRCs own MAP architecture. This is an ACS that shares the system memory between CPU and two accelerators FPGA via a system FPGA that implements a shared memory interconnect, which is called SNAP. This SNAP is directly connected into the DIMM sockets of the host system and provide a high bandwidth and low latency communication between FPGA and CPU. After using Xilinx FPGAs SRC switched with SRC-7 product line to Altera. Figure B.1 shows the diagram of the SRC-7 generation of `MAPstation`. 2015 they announced the `Saturn 1` server,

which is still an FPGA (Altera based) / Intel ACS system, that uses a second generation Carte compiler, called Carte++, for programming. No information on SRC Computers after 2016, their website and social media contacts are offline/closed, so the probability is high that they have gone out of business.

DESCRIPTION: According to a press release [47] Alteras FP Compiler is used. No further detailed information about the generated hardware is available. Carte used icc for the host code, Carte++ uses Clang/LLVM.

RESTRICTIONS: Unknown.

B.8 CASCADE

CLASSIFICATION: commercial, hwsw, automatic, dynamic

AUTHORS/COMPANY: CriticalBlue 2001. Since 2013 they focused on their products for software optimization and no longer offer their HLS tools.

REFERENCES: [51, 52]

TARGET: ARM and other RISC cores as host CPU. Structured ASIC, FPGA connected with AMBA AHB as accelerator.

DESCRIPTION: Similar to some other approaches, no hardware kernels are generated for acceleration, but instead a VLIW coprocessor is generated. The program parts that should be accelerate are translated into micro-code for that coprocessor. Also HW-SW interfacing functions are inserted and a C functional model and a test bench is generated.

RESTRICTIONS: No known restrictions.

B.9 CASH

OTHER NAMES/RELATED COMPILERS: Compiler for ASH, Project Phoenix

CLASSIFICATION: academic, dynamic, hwsw

AUTHORS/COMPANY: Carnegie Mellon University, Pittsburgh 2002

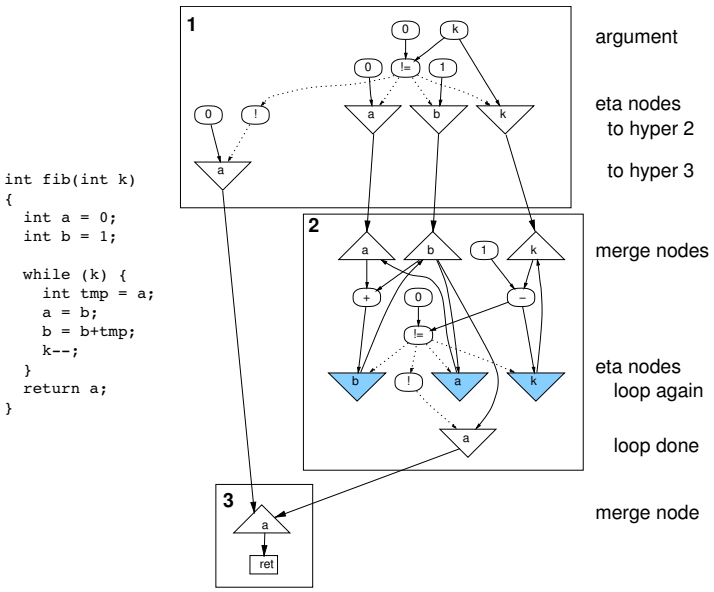


Figure B.2: A function in C computing the Fibonacci number and its Pegasus intermediate representation (from [26]).

REFERENCES: [26, 151]

TARGET: Asynchronous kernels with a supporting CPU for task that do not work well on reconfigurable hardware.

DESCRIPTION: Internal representation is a CMDFG in an IR called Pegasus (see Figure B.2 for an example). It uses a dynamic scheduling approach. Therefore, tokens flow from operation to operation activating them. Supports speculative execution with lenient operators.

RESTRICTIONS: The Verilog HDL backend does not support all feature the Pegasus IR provides. It lacks support for:

- Floating point
- Operator pipelining and pipeline balancing
- Function calls
- Loop unrolling
- Load Store Queue (LSQ)

As result of some missing optimizations the final speed compared with a 600 MHz CPU is a slow down up to a factor 4.5

B.10 CATAPULT-C

OTHER NAMES / RELATED COMPILERS: Catapult

CLASSIFICATION: commercial, static, hwsw, manual

AUTHORS / COMPANY: Mentor Graphics. Calypto Design Systems bought the compiler from Mentor Graphics in 2011. 2004

REFERENCES: [34, 132]

TARGET: RTL code for ASICs and FPGAs.

DESCRIPTION: Targets users who are hardware developers, to create more efficient RTL as with HDLs. Accepts C, C++ and SystemC as input.

RESTRICTIONS: Despite even allowing C++ input, no arbitrary pointers, no recursion, no `malloc()`, and just loops with compile-time limits are allowed. Can generate interfaces.

B.11 CHC

OTHER NAMES / RELATED COMPILERS: C-to-Hardware Compiler

CLASSIFICATION: commercial, static, hwsw, manual

AUTHORS / COMPANY: Altium 2008

REFERENCES: [8, 9]

TARGET: Creates (proprietary) hardware assembler output `HASM`, which can be converted into VHDL or Verilog.

DESCRIPTION: CHC generates a fixed synchronous schedule at compile-time. It implements several optimizations: unrolling, aggressive operator chaining, operator sharing, and it can optimize aliasing by considering the `restrict` keyword. Furthermore, it supports also some of the ISO TR18037 extension (embedded extension for C).

RESTRICTIONS: It does not support `setjmp` and `longjmp`. This is no unusual restriction, as it is, similar to recursion, not supported by almost all other compilers. But in this case,

even the code which remains in software is not allowed to contain those calls. Also no function pointer and var args are allowed in SW-HW transitions. Interfaces for the transitions can be automatically generated.

B.12 CHIMPS

OTHER NAMES / RELATED COMPILERS: Compiling High level language to Massively Pipelined System

CLASSIFICATION: academic, commercial, dynamic, hwsw

AUTHORS / COMPANY: University of Washington (Seattle) / Xilinx 2006

REFERENCES: [121, 123, 155, 156]

TARGET: Bee2 and Accelerated Compute Platform (ACP): Intel Xeon 7300 2.66 *GHz* with an Virtex 5 as RCU.

DESCRIPTION: CHiMPS sses a spatial data flow approach with dynamic scheduling. As interesting features it has a many-cache memory model. Memory accesses are not all done via a central cache, which is responsible for all acesses, but via many small caches, that are customized for the memory operation that access it. CHiMPS automatically creates and customizes those caches after it has analyzed the source code. Cache coherence is kept by a mechanism that orders dependent memory operations.

Like some other compilers it uses the LLVM [121] framework as front end. As additional IR, CHiMPS target language (CTL) is used, which resembles a simplified microprocessor instruction set.

RESTRICTIONS: Allows only one `return` statement, and `for` loops must follow the *single var init, simple condition, increment/decrement* pattern.

B.13 COSYMA

OTHER NAMES / RELATED COMPILERS: COSYnthesis of eMbedded Architectures, Braunschweig Synthesis System, BSS

CLASSIFICATION: academic, hwsw, automatic, static

AUTHORS/COMPANY: Technical University of Braunschweig 1994

REFERENCES: [63, 88]

TARGET: Target is a platform with a CPU for which application specific coprocessors are created. In the published papers this was a SPARC RISC processor, running with 33 MHz.

DESCRIPTION: One of the very few tools that automatically partitions the code into CPU and accelerator part. This is done via a cost model using simulated annealing. Input language is a C superset, called C^x, which adds instructions for time constraints, communication, processes and some user directives to the co–synthesis process to C. The CPU part is translated with gcc, while the part selected for the accelerator is output as CDFG and used as input for the BSS (Braunschweig Synthesis System) which synthesizes the hardware from it. BSS performs operator pipelining, loop pipelining and speculative computation with multiple branch prediction and schedules with a force directed scheduler.

RESTRICTIONS: Only very small code segments could be used as the resulting coprocessor runs with the same frequency as the CPU, and bigger code segments did not achieve the necessary timing.

B.14 CTOVERILOG

CLASSIFICATION: academic, static, hw

AUTHORS/COMPANY: Graduate project at the University of Haifa (Nadav Rotem), went online as `http://C-to-Verilog.com`, which is offline since 2015. 2009

REFERENCES: [16–19]

TARGET: FPGA and ASIC

DESCRIPTION: As many others this project uses the LLVM framework [121] as front end, and generates a static schedule with list scheduling. It performs some basic, hardware

oriented, optimizations, namely: loop unrolling, bitwidth reduction, arithmetic tree height reduction, and pipelining.

RESTRICTIONS: It is restricted to simple data flow and pointers to array locations are not permitted.

B.15 COMRADE

OTHER NAMES / RELATED COMPILERS: Comrade 2.0

CLASSIFICATION: academic, dynamic, hwsw, automatic

AUTHORS / COMPANY: TU Braunschweig, TU Darmstadt 2005

REFERENCES: [67, 107, 115]

TARGET: FPGA

DESCRIPTION: One of the few compilers that use dynamic scheduling. Supports HW-SW-coexecution and even allows to callbacks to software from the hardware (called software service). This compiler was the base for this work, so a full and detailed description is given at Section 3.3.

RESTRICTIONS: Does not support gotos and recursion.

B.16 CVC

CLASSIFICATION: commercial

AUTHORS / COMPANY: Hitachi 2008

REFERENCES: [104]

TARGET: FPGA

DESCRIPTION: This compiler does not take generic C as input, but just a small subset. This is no limitation, as it is only supposed to translate verification models coming from the CoMET5 co-verification environment.

RESTRICTIONS: It accepts no code with pointers, recursions, or loops. Due to its intended use, it is no restriction for the compiler, as the verification models do not contain code with those statements.

B.17 CYBER

OTHER NAMES / RELATED COMPILERS: CyberWorkbench, CWB

CLASSIFICATION: commercial, static

AUTHORS / COMPANY: NEC 1999

REFERENCES: [139, 189]

TARGET: SoC, FPGA

DESCRIPTION: As with many other compilers the intention is not to translate arbitrary C code, but to use C as hardware description language. They add statements for specification of in- and outputs, bitwidth, concurrency, synchronization, clocking, and data transfer types as registers, terminal, latch, and tri-state transfers. On the other hand, they forbid C code in common use to make it synthesizable.

Even the authors realized that it is no longer C-to-HDL translation, just C based hardware description, so they called their language BDL (Behaviour Description Language)

RESTRICTIONS: Supports no pointers, no recursion, and no dynamic memory.

B.18 DAEDALUS

OTHER NAMES / RELATED COMPILERS: PNgen

CLASSIFICATION: academic, dynamic, hwsw, automatic

AUTHORS / COMPANY: University of Amsterdam, Leiden University 2006

REFERENCES: [143, 144, 182, 186]

TARGET: Heterogeneous MultiProcessor-SoC

DESCRIPTION: Deadalus is a framework that generates a complete system. Part of it is PGgen, which generates Kahn-Process-Networks (KPN) [105] from C Code, which are turned into hardware.

RESTRICTIONS: Restricted to functions that consist of nested loops with data flow only.

B.19 DIME-C

CLASSIFICATION: commercial, static, hwsw

AUTHORS/COMPANY: Nallatech (now owned by Molex Inc.) 2007

REFERENCES: [29, 75]

TARGET: Designed to access Nallatech Hardware, e.g. Nallatech H101 cards. While the compiler generates for most computations generic VHDL code, memory interfaces and used library IP is specific to the Nallatech Hardware.

DESCRIPTION: Communication with FPGA memory or via FIFO has to be done by manually inserted library calls. When math operations should be realized in a pipelined manner, the regular C operators must be replaced by library calls. The remaining C based dialect was named DIME-C. Target HDL is VHDL.

RESTRICTIONS: The supported ANSI-C subset does not contain pointers, struct-arrays, and further misses some types of loops, and some types of switch statements.

B.20 EXCITE

OTHER NAMES/RELATED COMPILERS: NISC

CLASSIFICATION: academic, commercial, hwsw, manual, static

AUTHORS/COMPANY: Started as academic work at the University of Irvine. eXCite by YXI (Y Explorations, Inc.) is a commercial spin-off. 2005(NISC)/2006 (eXCite)

REFERENCES: [69, 85, 158, 202]

TARGET: ASIC and FPGA. For automatic interfacing generation only Altera is supported.

DESCRIPTION: NISC, meaning No-Instruction-Set-Computer, started as academic project at the University of California in Irvine. In the early version only hardware generation was performed, so no interfaces were generated. The commercial compiler eXCite is able to generate interfaces and provides even for some standard math library calls IP cores (like `sin()`, `sqrt()`). Partitioning must be performed manual on function level. It is able to integrate IP cores and perform hardware related optimizations like pipelining, bitwidth reduction and loop transformations.

RESTRICTIONS: The commercial compiler is very vague about the restrictions (their website states *"A substantial subset of ISO/ANSI C is supported"*), the NISC documentation is more specific, and states that it lacks support for function pointers, global pointer initialization, standard library calls, and long and double data types.

B.21 FP-COMPILER

OTHER NAMES/RELATED COMPILERS: Floating Point Compiler

CLASSIFICATION: commercial, data flow, static, hw

AUTHORS/COMPANY: Altera 2007

REFERENCES: [5, 120]

TARGET: FPGA

DESCRIPTION: Is supposed to translate pure data flow with floating point operations into VHDL. Due to its development state Altera has not integrated it into C2H. Later versions of C2H support floating point, so it can be assumed that it is now part of it. A press release announced that it is included in the Carte compiler[47].

RESTRICTIONS: Can only do data flow with floating point.

B.22 FPGA C

OTHER NAMES/RELATED COMPILERS: Transmogrifier C (tmcc)

CLASSIFICATION: academic (open source), static, manual, hwsw

AUTHORS/COMPANY: Started as Transmogrifier C at the University of Toronto, and was later continued as open source project *FPGA C*. 1994 (tmcc),2005 (*FPGA C*)

REFERENCES: [14, 71, 72, 124]

TARGET: Transmogrifier C targets a series of FPGA systems built in the nineties at the university of Toronto that were called Transmogrifier. The generated circuits could be used for other FPGAs, CPLD, or ASICs, too.

DESCRIPTION: Due to the many limitations (see below), even the authors admit that it is more a C-based HDL than a compiler translating arbitrary C to hardware. The scheduling is static and the compiler relies for the interfacing logic on manual insertion. It translates control flow into data flow (via multiplexers) and can efficiently translate functions with at most 4 inputs. FPGA C has not improved much, gone is just the restriction with non `int` sized data types (`short`, `long`, structs and small arrays) and `for` loops.

RESTRICTIONS: Transmogrifier was limited and did not support the following operators and statements: Strings, casts, multiply, divide, arrays, pointers, structures, `for` loops, floating point, `sizeof`, `continue`, `goto`, `do-while` loop, `&` and * operator `/=`, `switch-case`, `short`, `long`.

B.23 GARPCC

CLASSIFICATION: academic, static, hwsw, automatic

AUTHORS/COMPANY: University of California at Berkeley 2000

REFERENCES: [31–33, 87]

TARGET: Target is the Garp architecture (a MIPS processor with a reconfigurable coprocessor).

DESCRIPTION: This compiler (using a SUIF front end) is one of the few that partitions the program automatically in HW

and SW. Profiling with a sample data set the execution count is used as criteria for hardware extraction.

This compiler was followed by the Nimble (see Appendix B.32) project.

RESTRICTIONS: Does not support function calls, recursion, gotos. Only supports data flow with control flow that can be mapped to predicated execution.

B.24 GAUT

CLASSIFICATION: academic, manual, static

AUTHORS / COMPANY: Université Bretagne Sud (UBS), France 2005

REFERENCES: [70]

TARGET: Output is generic VHDL (with focus on DSP applications).

DESCRIPTION: The C code is parsed in the newest version of GAUT with gcc and a plugin is used to pass the CDFG to the GAUT tool.

RESTRICTIONS: GAUT does not support variable-bound loops, gotos, and pointer and array usage is very limited (they can only be used with either read or write accesses, not both on the same variable, and the address must be known at compile time).

B.25 GCC2VERILOG

CLASSIFICATION: academic, static, manual, hwsw

AUTHORS / COMPANY: Korea University, Seoul 2010

REFERENCES: [91]

TARGET: Generic VHDL. For the HWSW coexecution a PICO microprocessor is used.

DESCRIPTION: Partitioning must be performed manually with function-granularity. Research yielded no applications implemented with this compiler, the claimed performance gain is questionable. Not only that the designs are not pipelined, bigger examples would require a lot of routing resources to the memory ports (the author already acknowledges that in the future work section, the memory accesses must be further optimized).

RESTRICTIONS: No restrictions. Pointers can not point to memory ranges that have an unknown size at compile time.

B.26 HANDEL-C

CLASSIFICATION: academic, static, commercial, manual

AUTHORS/COMPANY: Introduced by Embedded Solutions Limited (ESL) as spin-off from Oxford University. In 2000 ESL was renamed Celoxica, and 2008 aquired by Agility. 2009 acquired by Mentor Graphics. 1996

REFERENCES: [24]

TARGET: FPGAs and PLDs.

DESCRIPTION: Using extensions of the C language, called Handel-C, the user is required to schedule and model parallelism himself. This and the restrictions make this dialect to a HDL C-like syntax.

RESTRICTIONS: It explicit lacks support for ANSI-C constructs which are not appropriate for hardware implementation, that includes: no pointers, only compile-time array indices (except for special RAM/ROM-types), no expression with side effects (e.g. `b=a++` is forbidden), no type conversion, user must schedule the operations, and no floating-point variables (only with explicit use of a floating-point library). But instead additional constructs (like streams and channels for communication or instructions to express parallelism) for hardware design are available.

B.27 HTHREADS

OTHER NAMES/RELATED COMPILERS: C2VHDL, HybridThreads Compiler

CLASSIFICATION: academic, hwsw, manual, static

AUTHORS/COMPANY: University of Kansas 2005, development of the HybridThreads compiler was halted 2009.

REFERENCES: [11]

TARGET: FPGA

DESCRIPTION: Hthreads is just the framework to manage and schedule multiple threads on an FPGA Hardware. Part of it is a compiler, the HybridThreads Compiler, that can generate VHDL code from C. The compiler uses gcc as front end, and converts the gcc internal GIMPLE IR into an IR closer to hardware called *HIF* (Hardware Intermediate Format). The HybridThreads Compiler is already the second generation compiler and the development was halted before it was finished. The previous compiler was called C2VHDL and was very limited in its function and used *C–* as IR.

RESTRICTIONS: C2VHDL allows no function calls, no memory accesses, no division and modulo operation and no floating point operations.

B.28 IMPULSE-C

OTHER NAMES/RELATED COMPILERS: Streams-C

CLASSIFICATION: academic, commercial, manual, hwsw, static

AUTHORS/COMPANY: Los Alamos National Laboratory, licensed to Impulse Accelerated Technologies 2003 (Streams-C 2000)

REFERENCES: [65, 77, 99]

TARGET: FPGA

DESCRIPTION: Streams-C uses the infrastructure from the Napa C compiler (see Appendix B.31). As additional functionality stream handling was added, which is based on a CSP approach.

RESTRICTIONS: Only limited pointer support (they must resolve at compile time to array references), very limited `struct` support, limited control flow (loops and predicates), no software calls, and no recursion.

B.29 LEGUP

CLASSIFICATION: Academic, HWSW, static, automatic

AUTHORS / COMPANY: University of Toronto 2010

REFERENCES: [35–37, 83]

TARGET: FPGA with a 32 bit soft core MIPS

DESCRIPTION: LegUp uses the LLVM framework. It can not only translate regular C code, but can use pthreads and OpenMP instructions for parallelism to generate parallel hardware units. The operations are scheduled with an ASAP scheduler. The decision which parts should be accelerated with the FPGA hardware is based on profiling runs, with a granularity of functions.

RESTRICTIONS: Dynamic memory, floating point, functions returning structs and recursions are not supported.

B.30 MITRION-C

CLASSIFICATION: Commercial, dynamic, hwsw

AUTHORS / COMPANY: Mitrionics 2005

REFERENCES: [134]

TARGET: FPGA

DESCRIPTION: This is just a C-like language, which requires explicit description of many hardware structures. The compiler then generates no application specific hardware, but custom processors (Mitrion Virtual Processors) with code for them.

RESTRICTIONS: Mitrion-C does not support pointers, gotos, recursion, and requires that every variable is allowed to be written just once in each scope.

B.31 NAPA C

OTHER NAMES/RELATED COMPILERS: MARGE (Malleable Architecture Generator)

CLASSIFICATION: Academic, commercial manual, static, hwsw

AUTHORS/COMPANY: Sarnoff Corporation (now SRI International) 1997

REFERENCES: [76, 78, 160]

TARGET: National Semiconductors NAPA1000 chip. This is one of the first products that actually pairs up a CPU (in this case a small 32 bit RISC processor, called FIP (for Fixed Instruction Processor) together with a reconfigurable unit ALP (Adaptive Logic Processor) in a single chip.

DESCRIPTION: Input for the compiler, which uses the SUIF framework [84, 168], is C code, with annotated pragmas. The pragmas are responsible for the partitioning and declare the storage (memory/registers) for variables. After partitioning the following steps are done by a NSC proprietary Compiler for the FIP and by the pure data path compiler MARGE [78] for the ALP.

RESTRICTIONS: The ALP can only execute data path constructs. Any control flow must be handled by the FIP.

B.32 NIMBLE

CLASSIFICATION: Academic, static, hwsw, automatic

AUTHORS/COMPANY: University of California at Berkeley / Synopsys 1998

REFERENCES: [125]

TARGET: ACE-V platform (Xilinx Virtex with Sun MicroSparc II processor). Retargetable with ADL (Architecture Description Language)

DESCRIPTION: Based on the GarpCC (see Appendix B.23) compiler.

RESTRICTIONS: Nimble does not support gotos, recursion and floating point operations.

B.33 NYMBLE

CLASSIFICATION: Academic, static, hwsw, manual

AUTHORS / COMPANY: Technische Universität Darmstadt 2010

REFERENCES: [93, 95, 97]

TARGET: FPGA

DESCRIPTION: Early versions used the Scale framework, while later use the LLVM framework.

RESTRICTIONS: The compiler supports no recursion and no gotos.

B.34 BAMBU

CLASSIFICATION: academic, hwsw, automatic, static

AUTHORS / COMPANY: Politechnico di Milano 2012

REFERENCES: FPGA

TARGET: [152]

DESCRIPTION: This compiler is part of the Panda project, which aims to be framework in all aspects of the HW-SW Co-design field, and is a successor to the hArtes (see Appendix B.41) project. It uses the same partioning tool *Zebu*, but a new compiler, Bambu.

Bambu is GCC based.

RESTRICTIONS: Does not support gotos and recursion.

B.35 PICO-EXPRESS

OTHER NAMES / RELATED COMPILERS: Program In Chip Out, Synphony C (ab 2010)

CLASSIFICATION: Commercial

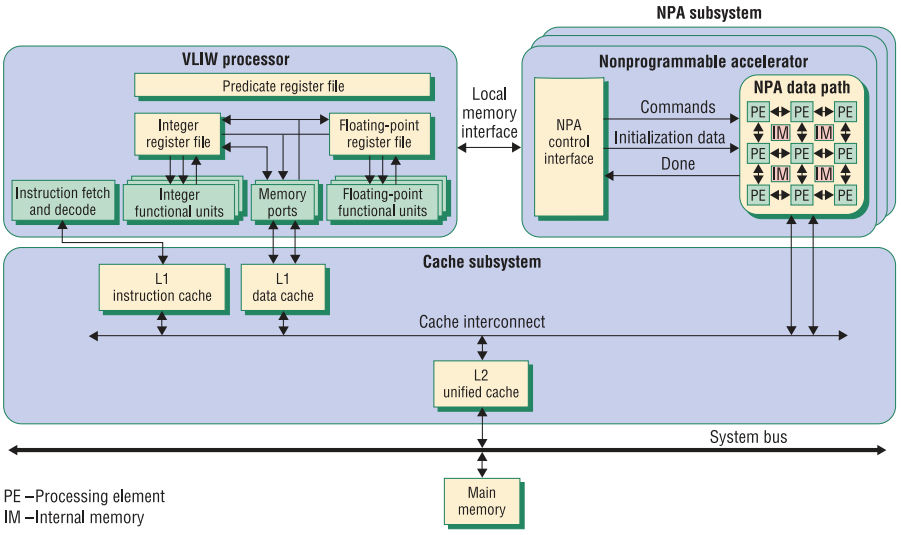


Figure B.3: PICO system-level architecture template (from [108]).

AUTHORS / COMPANY: (HP Labs spin off) Synfora, later Synopsys

2002

REFERENCES: Application Specific Computer Systems

TARGET: [108]

DESCRIPTION: An architectural template (see Figure B.3) is used to generate an application specific system. The system contains a custom VLIW cpu to execute parts than can not be accelerated. Parts than can be accelerated become a dedicated nonprogrammable accelerator.

RESTRICTIONS: No floating point, no &-operator, no pointer arithmetic, no unions, no goto, no recursion, no static variables, no library functions, only limited nesting (it is forbidden to have two sequential loops enclosed within another loop).

B.36 PRISC

OTHER NAMES / RELATED COMPILERS: PRogrammable Instruction Set Computers

CLASSIFICATION: Academic

AUTHORS / COMPANY: Harvard University, Cambridge, Digital Equipment Corporation

1994

REFERENCES: [157]

TARGET: Target is a (not existing) CPU which has small hardware-programmable resources, called PFU, in its datapath. The PFU has access to the register file of the CPU and is supposed to have a latency of one CPU cycle.

DESCRIPTION: Due to the very small amount of logic resources, and the fact that all operations must be performed in at most one cycle, the acceleration is limited to very small code portions.

RESTRICTIONS: Supports no control flow.

B.37 ROCCC (RIVERSIDE OPTIMIZING COMPILER FOR CONFIGURABLE COMPUTING)

OTHER NAMES / RELATED COMPILERS: ROCCC 2.0, SA-C

CLASSIFICATION: Academic, static, hw, manual

AUTHORS / COMPANY: University of California, Riverside 2008

REFERENCES: [28, 79, 187]

TARGET: Generic VHDL output for the kernel

DESCRIPTION: ROCCC is the successor of the SA-C compiler

RESTRICTIONS: Memory addresses must be non at compile time, no pointers, several restrictions regarding the loop variables (compile time constant stride, no access out of loop header)

B.38 SPARK

CLASSIFICATION: academic, static

AUTHORS / COMPANY: University of California, Irvine, funded by SRC and Intel 2003

REFERENCES: [80], [81]

TARGET: VHDL

DESCRIPTION: As IR are hierarchical task graphs used. Data paths are scheduled with a priority based list scheduler.

RESTRICTIONS: Does not support pointers and recursion

B.39 TRIDENT

OTHER NAMES/RELATED COMPILERS: Sea Cucumber Compiler

CLASSIFICATION: OpenSource, static

AUTHORS/COMPANY: Los Alamos National Laboratory 2005

REFERENCES: [183, 184]

TARGET: Generic VHDL

DESCRIPTION: Trident builds on and shares code from the Sea Cucumber compiler, which is a Java-to-HDL compiler. It uses the LLVM framework for parsing and optimizations and transforms LLVMs byte code into a Trident specific IR. The loops are modulo scheduled, the rest is scheduled force-directed.

RESTRICTIONS: Trident lacks support for pointers and memory accesses, gotos and recursion.

B.40 XPP-VC

CLASSIFICATION: Commercial, dynamic, hwsw, manual

AUTHORS/COMPANY: PACT XPP Technologies AG 2002

REFERENCES: [39]

TARGET: XPP-III, a coarse granular reconfigurable array with a VLIW-like CPU.

DESCRIPTION: The XPP-VC compiler uses the SUIF framework. It can translate the selected C code into configuration data for the coarse grained XPP array. As the available array resources may be not enough, the compiler uses temporal partitioning.

RESTRICTIONS: The C code for the reconfigurable array is not allowed to contain struct and floating-point data types, pointers, irregular control flow, recursion and system calls.

B.41 DWARV

OTHER NAMES / RELATED COMPILERS: Delft Workbench Automated Reconfigurable VHDL Generator, Molen

CLASSIFICATION: Academic, hwsw, static, dynamic, manual

AUTHORS / COMPANY: Delft University of Technology 2007

REFERENCES: [20, 145, 172, 179]

TARGET: Targeting the Molen platform. A prototype version is implemented on a Virtex II Pro board using a PowerPC 405 as CPU.

DESCRIPTION: The Molen compiler framework contains a compiler which is not able to generate hardware itself, but to generate the necessary interfaces for the marked functions in software and to embed the hardware bitstreams for them into the generated binary.

The hardware which is inserted must be generated with a different software. In the Molen publications two ways are mentioned: One is the DWARV compiler. This SUIF2 based compiler can translate a pure DFG into VHDL, which can be synthesized into static scheduled hardware. The other approach makes use of the tools Compaan and Laura. Here some more manual interaction was necessary, as the translated function must be first translated into Matlab Code. This got turned via Compaan into a KPN, which got turned via Laura into dynamic scheduled hardware.

The DWARV compiler is also used in the hArtes framework. There the partition happens automatically with a tool called *Zebu*.

RESTRICTIONS: DWARV supports data flow only.

C

TRIAD SYNTAX

C.1 EBNF

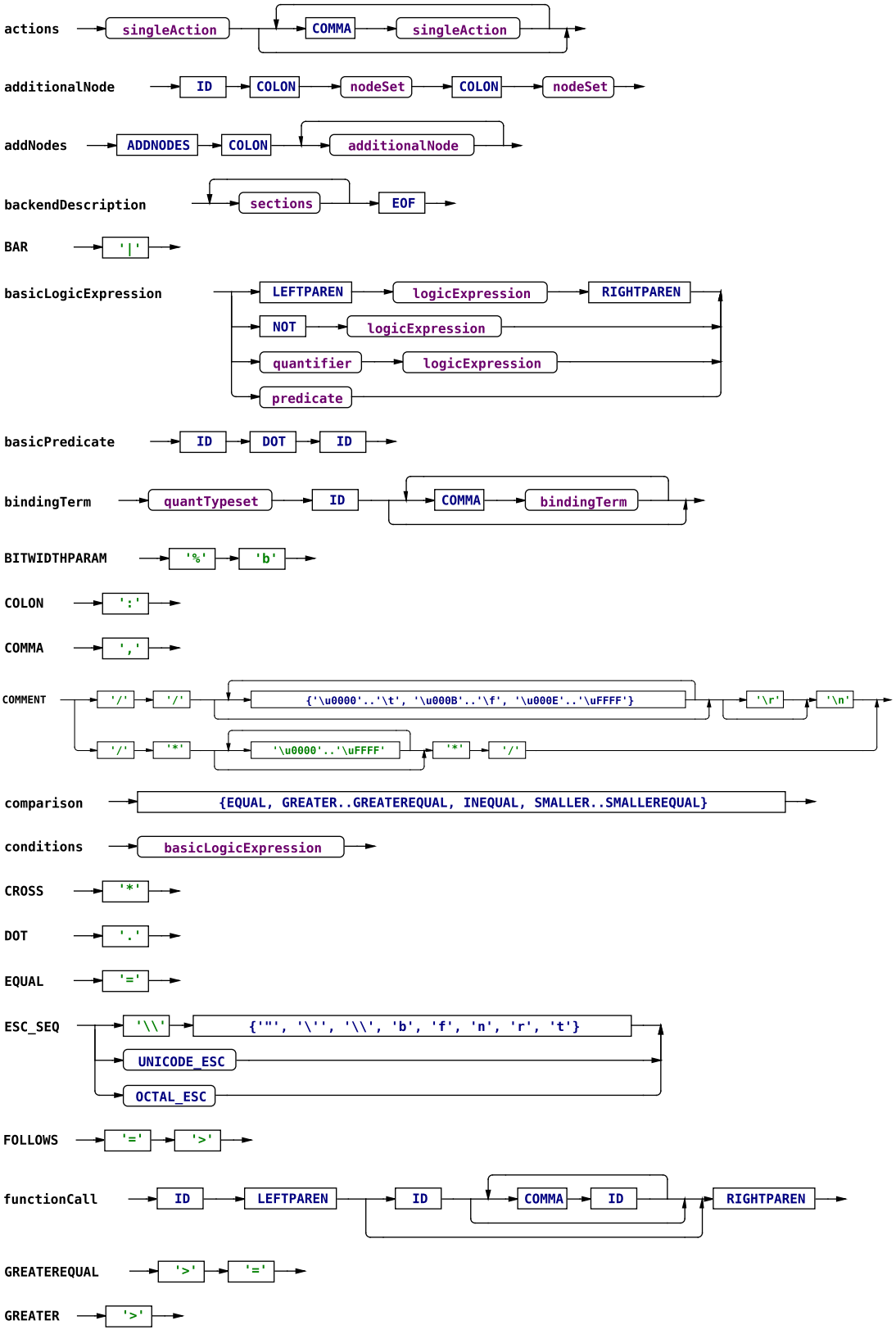
```
backendDescription
         ::= sections+ EOF
sections ::= hwOpsListSection
           | hwOpsMapSection
           | schedulingDecl
           | addNodes
           | tokensDecl tokenRules
hwOpsMapSection
         ::= 'MAPPING' ':' hwOpsMap
hwOpsMap ::= hwOpMapping*
hwOpMapping
         ::= ID '(' typeToHwOps ')'
typeName ::= 'UINT'
           | 'SINT'
           | 'FLOAT'
           | 'DOUBLE'
logicalResult
         ::= 'LOG'
           | 'ARITH'
typeToHwOps
         ::= typeName ( ',' typeName )* ':' ID ( '('
             hwOpsParameters ')' '(' unconnectedPorts? ')' )
             ?
unconnectedPorts
         ::= ID ( ',' ID )*
hwOpsParameters
         ::= hwOpParameter ( ',' hwOpParameter )*
hwOpParameter
         ::= ID '=' hwOpParameterValue
hwOpParameterValue
         ::= INT
           | '%s'
           | '%b'
schedulingDecl
         ::= 'SCHEDULING' ':' ( 'STATIC' | 'DYNAMIC' )
hwOpsListSection
```

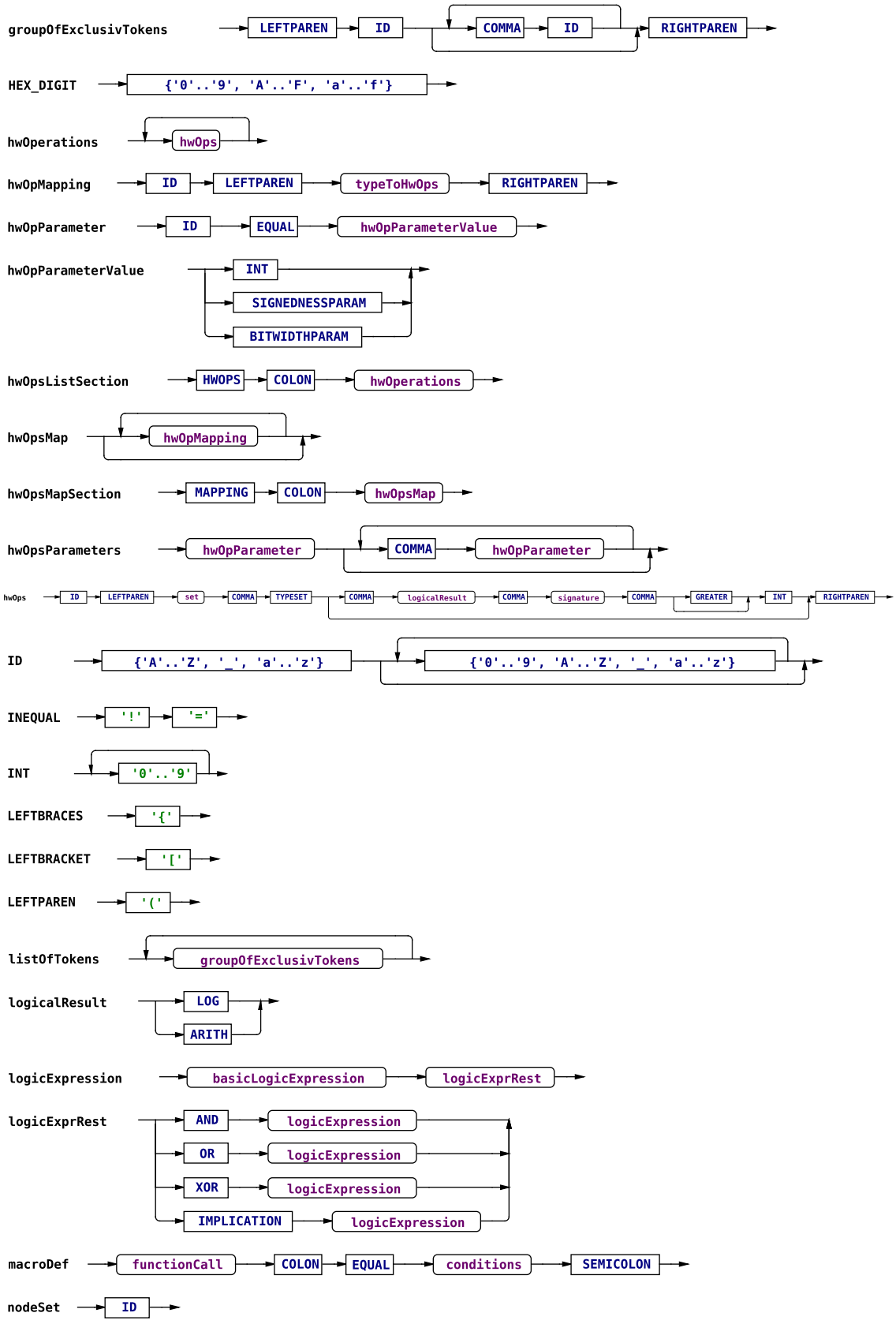
```
         ::= 'HWOPS' ':' hwOperations
hwOperations
         ::= hwOps+
hwOps    ::= ID '(' set ',' TYPESET ( ',' logicalResult ','
    signature ',' '>'? INT )? ')'
signature
         ::= ID ( '*' ID )* '=>' ID ( '*' ID )*
set      ::= '[' numberRange ( ',' numberRange )* ']'
numberRange
         ::= INT
           | INT '-' INT
tokensDecl
         ::= 'TOKENS' ':' listOfTokens
listOfTokens
         ::= groupOfExclusivTokens+
groupOfExclusivTokens
         ::= '(' ID ( ',' ID )* ')'
tokenRules
         ::= 'RULES' ':' ( tokenRule | macroDef )*
macroDef ::= functionCall ':' '=' conditions ';'
nodeSet  ::= ID
tokenRule
         ::= '{' unboundVariables '|' conditions '}' '=>'
             actions ';'
conditions
         ::= basicLogicExpression
unboundVariables
         ::= singleUnboundVariable ( ','
             singleUnboundVariable )*
singleUnboundVariable
         ::= ID ID
actions  ::= singleAction ( ',' singleAction )*
singleAction
         ::= functionCall
functionCall
         ::= ID '(' ( ID ( ',' ID )* )? ')'
basicLogicExpression
         ::= '(' logicExpression ')'
           | '!' logicExpression
           | quantifier logicExpression
           | predicate
logicExpression
         ::= basicLogicExpression logicExprRest
logicExprRest
         ::= '$\wedge$' logicExpression
           | '$\vee$' logicExpression
```

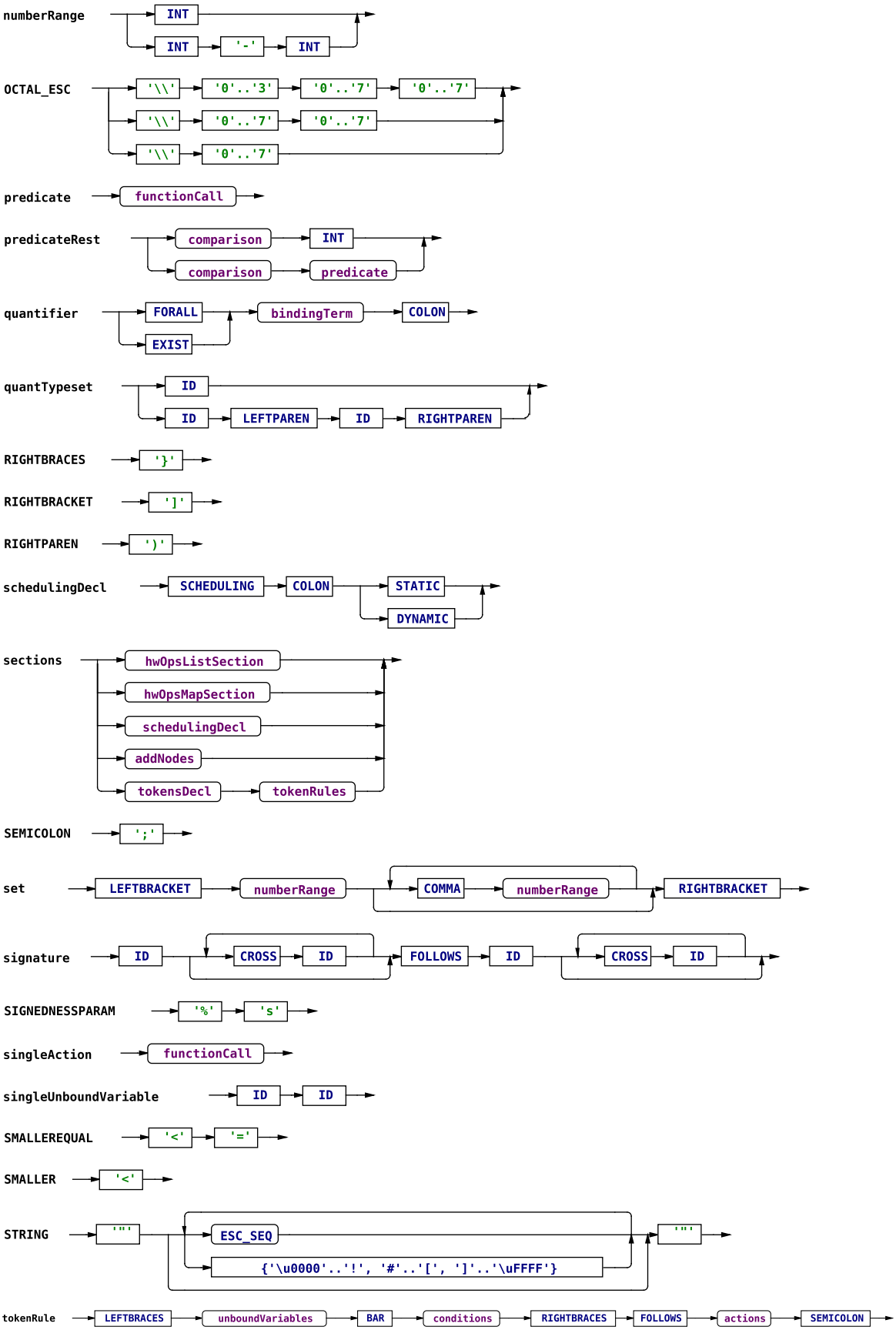
```
           | '$\otimes$' logicExpression
           | '$\rightarrow$' logicExpression
quantifier
         ::= ( '$\forall$' | '$\exists$' ) bindingTerm ':'
bindingTerm
         ::= quantTypeset ID ( ',' bindingTerm )*
quantTypeset
         ::= ID
           | ID '(' ID ')'
basicPredicate
         ::= ID '.' ID
predicate
         ::= functionCall
comparison
         ::= '<'
           | '>'
           | '<='
           | '>='
           | '='
           | '!='
addNodes ::= 'NODES' ':' additionalNode+
additionalNode
         ::= ID ':' nodeSet ':' nodeSet
_        ::= COMMENT
           | WS
           /* ws: definition */

<?TOKENS?>

TYPESET  ::= 'u'
           | 's'
           | 'f'
           | 'x'
ID       ::= ( [a-z] | [A-Z] | '_' ) ( [a-z] | [A-Z] | [0-9]
     | '_' )*
INT      ::= [0-9]+
COMMENT? ::= '//' [^#xA#xD]* #xD? #xA
           | '/*' .* '*/'
WS       ::= ' '
           | #x9
           | #xD
           | #xA
STRING   ::= '"' ( ESC_SEQ | [^\"] )* '"'
HEX_DIGIT
         ::= [0-9]
           | [a-f]
```

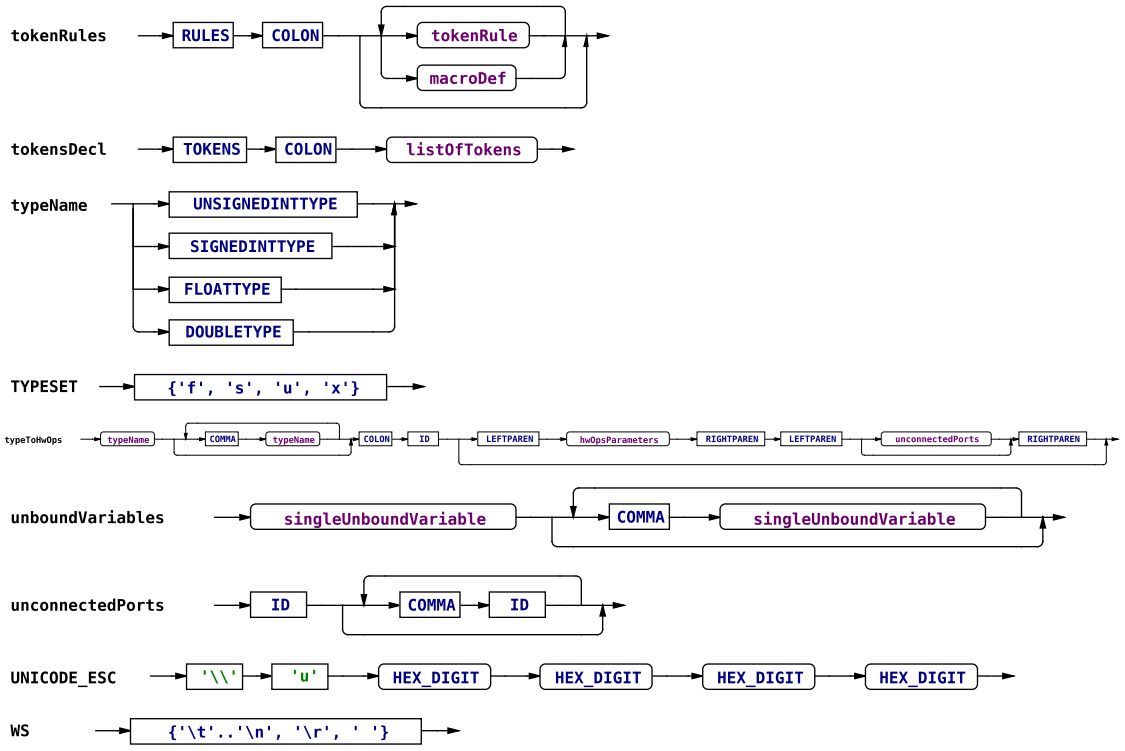
```
           | [A-F]
ESC_SEQ  ::= '\' ( 'b' | 't' | 'n' | 'f' | 'r' | '"' | "'" |
     '\' )
           | UNICODE_ESC
           | OCTAL_ESC
OCTAL_ESC
         ::= '\' [0-3] [0-7] [0-7]
           | '\' [0-7] [0-7]
           | '\' [0-7]
UNICODE_ESC
         ::= '\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
EOF      ::= $
```

C.2 SYNTAX DIAGRAMS




groupOfExclusivTokens
LEFTPAREN
ID
COMMA
ID
RIGHTPAREN
HEX_DIGIT
{'0'..'9', 'A'..'F', 'a'..'f'}
hwOperations
hwOps
hwOpMapping
ID
LEFTPAREN
typeToHwOps
RIGHTPAREN
hwOpParameter
ID
EQUAL
hwOpParameterValue
hwOpParameterValue
INT
SIGNEDNESSPARAM
BITWIDTHPARAM
hwOpsListSection
HWOPS
COLON
hwOperations
hwOpsMap
hwOpMapping
hwOpsMapSection
MAPPING
COLON
hwOpsMap
hwOpsParameters
hwOpParameter
COMMA
hwOpParameter
hwOps
ID
LEFTPAREN
set
COMMA
TYPESET
COMMA
logicalResult
COMMA
signature
COMMA
GREATER
INT
RIGHTPAREN
ID
{'A'..'Z', '_', 'a'..'z'}
{'0'..'9', 'A'..'Z', '_', 'a'..'z'}
INEQUAL
'!'
'='
INT
'0'..'9'
LEFTBRACES
'{'
LEFTBRACKET
'['
LEFTPAREN
'('
listOfTokens
groupOfExclusivTokens
logicalResult
LOG
ARITH
logicExpression
basicLogicExpression
logicExprRest
logicExprRest
AND
logicExpression
OR
logicExpression
XOR
logicExpression
IMPLICATION
logicExpression
macroDef
functionCall
COLON
EQUAL
conditions
SEMICOLON
nodeSet
ID


numberRange
INT
INT
'-'
INT
OCTAL_ESC
'\\'
'0'..'3'
'0'..'7'
'0'..'7'
'\\'
'0'..'7'
'0'..'7'
'\\'
'0'..'7'
predicate
functionCall
predicateRest
comparison
INT
comparison
predicate
quantifier
FORALL
EXIST
bindingTerm
COLON
quantTypeset
ID
ID
LEFTPAREN
ID
RIGHTPAREN
RIGHTBRACES
'}'
RIGHTBRACKET
']'
RIGHTPAREN
')'
schedulingDecl
SCHEDULING
COLON
STATIC
DYNAMIC
sections
hwOpsListSection
hwOpsMapSection
schedulingDecl
addNodes
tokensDecl
tokenRules
SEMICOLON
';'
set
LEFTBRACKET
numberRange
COMMA
numberRange
RIGHTBRACKET
signature
ID
CROSS
ID
FOLLOWS
ID
CROSS
ID
SIGNEDNESSPARAM
'%'
's'
singleAction
functionCall
singleUnboundVariable
ID
ID
SMALLEREQUAL
'<'
'='
SMALLER
'<'
STRING
'"'
ESC_SEQ
{'\u0000'..'!', '#'..'[', ']'..'\uFFFF'}
'"'
tokenRule
LEFTBRACES
unboundVariables
BAR
conditions
RIGHTBRACES
FOLLOWS
actions
SEMICOLON


tokenRules
RULES
COLON
tokenRule
macroDef
tokensDecl
TOKENS
COLON
listOfTokens
typeName
UNSIGNEDINTTYPE
SIGNEDINTTYPE
FLOATTYPE
DOUBLETYPE
TYPESET
{'f', 's', 'u', 'x'}
typeToHwOps
typeName
COMMA
typeName
COLON
ID
LEFTPAREN
hwOpsParameters
RIGHTPAREN
LEFTPAREN
unconnectedPorts
RIGHTPAREN
unboundVariables
singleUnboundVariable
COMMA
singleUnboundVariable
unconnectedPorts
ID
COMMA
ID
UNICODE_ESC
'\\'
'u'
HEX_DIGIT
HEX_DIGIT
HEX_DIGIT
HEX_DIGIT
WS
{'\t'..'\n', '\r', ' '}

D

TRIAD FILE

D.1 *COCOMA* DYNAMIC *CANCEL* TOKENS RULES

Listing D.1: *Triad* rule set for *COCOMA* (using dynamic *Cancel* tokens).

```
TOKENS: (*Activate*)(Cancel)(NotEnterable)(MemoryAccess)

RULES:

// rules are labeled according to the cases in Figure
    5.1/5.2

//(*) (a) standard activation of an operator node
// when all inputs of an operation have a token, and the
    output is none, start the operation

{ Node node | (∀ Inputs(node) input:( hasToken(input,
    Activate)
                                    ∧ !hasToken(input,
                                        Cancel) )  // from (
                                        b)
                 )
              ∧ !isRunning(node)
              ∧ !∃ Outputs(node) output: hasToken(output
                  Activate)
              ∧ !isMemOps(node)                 // from (e)
             }
     =>  start(node),
         delete(input, Activate);

// when an operation is finished, create activate token at
    output
{ Node node, Output o | isFinished(node) ∧ isOutputOf(o,
    node)
                      }
     =>  create( o, Activate);

// forwarding from one operator to the next
//(*)defining helper macro
```

```
allSuccsWithoutToken(output) := ∀ Succs(output) ins: (!
    hasToken(ins, Activate)
                                    ∧ !hasToken(ins, Cancel
                                        ) ; // from (b)
{ Node node1 | ∃ Outputs(node1) output, Inputs(node2)
    input:
                ( allSuccsWithoutToken(output)
                ∧ hasToken(output, Activate) ∧ isSucc(node2
                    , node1) )}

     => delete(output, Activate),
        create(input, Activate);


// (b) cancel token cancels out activate token

 // (*)if there is one AT at one of the inputs delete it
 { Node node | (∀ Outputs(node) output: hasToken(output,
     Cancel))
                ∧ ∃ Inputs(node) input: hasToken(input,
                    Activate)}
      => delete(input, Cancel)
         delete(output, Activate)
         reset(node);

 // if not place it at the inputs
 { (*)Node node | (∀ Outputs(node) output: hasToken(output
     , Cancel))
                ∧ ∃ Inputs(node) input: !hasToken(input,
                    Activate)}
      => create(input, Cancel)
         delete(output, Activate)
         reset(node);

 // (*)CT flow upwards (if already AT coming, we wait for
     it, else we send it)
 { Node node, Input i | (∀Preds(i) output: !hasToken(
     output, Activate))
                ∧ (isInputOf(i, node) ∧ hasToken(input,
                    Cancel)}
      => delete(input, Cancel)
         create(output, Cancel);

  // at the input place AT and CT cancel out
  {Input i | hasToken(i, Activate) ∧ hasToken(i, Cancel)}
      => delete(i, Activate)
```

```
        delete(i, Cancel);

// (c) create cancel tokens at multiplexers
  { Node node, Input select | isJoinNode(node) ∧
     isInputPortOf(select, node) ∧ isSelectInput(select)
     // mux
              ∧ hasToken(select, Activate)
              ∧ ( ∃ Inputs(node) input: (hasToken(input,
                  Activate) ∧ !isSelectInput(input))) //
                  if one other input has AT
              ∧ ( ∃ Inputs(node) nonAct: (!hasToken(input
                  , Activate))) // term required to select
                   the inputs w/o AT
              ∧ ( ∃ Outputs(node) output: !hasToken(
                  output, Activate)) // when already AT at
                   output, do nothing (stall)
              ∧ !isLoopInitMux(node) // from (d)
              }
     => create(nonAct, Cancel)
        delete(select, Activate)
        create(output, Activate);

// (d) token flow at loop multiplexer nodes,
//     entereable State is modelled with NotEnterable token
     at the node
//    (inverted, because token is default not present)
  { Node n | isLoopInitMux(n)
           ∧ (∃ Inputs(n) select: (isSelectInput(select,
                n) ∧ hasToken(select,Activate)))
           ∧ !hasToken(n, notEnterable)
           ∧ (∃ Inputs(n) init: (isInitInput(init, n) ∧
                hasToken(init, Activate)))
         }
     => create(n, NotEnterable)
        delete(init, Activate)
        create(output, Activate);

// resetting internal state when CT comes in
  { Node n | isLoopInitMux(n)
            ∧ (∃ Outputs(n) o: hasToken(o,Cancel))
            ∧ hasToken(n, notEnterable)
          }
     => delete(n, NotEnterable)
        delete(o, Cancel);

// (e) token forwarding along memory edges
```

```
  // activiation of mem ops
  { Memops m, Memops p | (∀ Inputs(m) input:( hasToken(
      input, Activate)
                        ∧ !hasToken(input, Cancel) ) //
                            from (b)
                        )
                        ∧ !isRunning(m)
                        ∧ !∃ Outputs(m) output: hasToken(
                            output,Activate)
                        ∧ isImmedMemoryPred(p, m) ∧
                            hasToken(p, MemoryAccess) //
                            prev. gives Mem token
                        ∧ ∀ Nodes cond: (isControlledBy(m
                            , cond) → (
                            isControlledAndTrue(m, cond) ∧
                             ∀ Outputs(cond) o: hasToken(o
                            , Activate))) // forbid
                            speculation of memops
                        }
      =>  start(m),
          delete(input, Activate)
          delete(m, MemoryAccess);

   // creating mem token
   { Memops m, Output o| isFinished(m) ∧ isOutputOf(o, m)}
       =>  create( m2, MemoryAccess);  // no AT necessary,
           generated by (a) rule

  // (f) token creadtion at a special (all) token node,
      covered by the other rules implementation
  // (g) token creadtion at a special (all) token node,
      covered by the other rules implementation


  // (h) -(j) rules for condition nodes

  // (h) activate token creation along control edges

  //(*) (i) control edges creates cancel token along data
      edge


  // (j) cancel token creation along control edges

   // remaining cases refer to annotated edges
```

```
// edges with annotations cover special cases, which are
    handled in the other rules
// (k) handling Tokens for edges with "nCT" (= no cancel
    token) annotation

// (l) handling Tokens for edges with "nAT" (= no activate
    token)  annotation

// (m) handling Tokens for edges with "alwActAT" (= always
    activate token)  annotation, when activated via control
     flow

// (n) handling Tokens for edges with "alwActAT" (= always
    activate token)  annotation, when not activated via
    control flow


// (o)+(p) used for loop multiplexer
// (o) handling AT  for edges with "atOnCancel" (= creates
    AT if cond. is false)  annotation
// (p) handling CT  for edges with "atOnCancel" (= creates
    AT if cond. is false)  annotation


// activate constant nodes
{ Node n, Output o: isConstant(n) ∧ isOutputPort(o, n)}
    => create(o, Activate);
```

D.2 COCOMA STATIC CANCEL TOKENS RULES

To show, how small the changes to the original rule set from Appendix D.1 are, and for better comparability, they are just described here:

- Remove all *create*(..., *Cancel*), *delete*(..., *Cancel*) and all terms with *hasToken*(..., *Cancel*) in rules marked with a `(*)` in the comments.
- Add the folling rules (Sending CT to memory nodes, and canceling there):

```
{ MemOps m, Node cond | isControlledByAndFalse(m,
    cond)∧∃
                          Outputs(cond): hasToken(
                              cond,
                        Activate)∧∃
                          Inputs(m) i: true)
```

```
     => create(i, Cancel);


{ MemOps m, Outputs o | ∀ Inputs(m) i: hasToken(i,
    Activate)∧
                        hasToken(m, Cancel); ∧
                            isOutputOf(o, m)}
     => create(o, Activate)
        delete(i, Activate)
        delete(m, Cancel);
```

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Sumit Ahuja, Swathi T. Gurumani, Chad Spackman, and Sandeep K. Shukla. "Hardware Coprocessor Synthesis from an ANSI C Specification." In: *IEEE Des. Test* 26 (4 July 2009), pp. 58–67. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.81. URL: http://dl.acm.org/citation.cfm?id=1608567.1608653.

[3] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzaki. *An Overview of the SUIF2 Compiler Infrastructure*. Tech. rep. Stanford, CA, USA: Stanford University, 2000.

[4] Abdul Alshadadi, David Marx, Johannes Meyer, and Rosten Sillus. "Bachelorpraktikum – Graphenvisualisierung GAP." Comp. software. Technische Universität Darmstadt, 2009.

[5] Altera. *Altera Whitepaper – Floating-Point Compiler – Increasing Performance With Viewer Resources*. Altera Corporation. 2007. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01050-floating-point-compiler-increasing-performance-with-fewer-resources.pdf.

[6] Altera. *Nios II C2H Compiler User Guide*. Altera Corporation. 2009. URL: http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf.

[7] Altera. *Altera SDK for OpenCL*. Altera Corporation. 2012. URL: https://www.altera.com/ja_JP/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[8] Altium Ltd. *Getting Started with the C-to-Hardware Compiler*. 19th May 2008. URL: http://www.altium.com/files/altiumdesigner/s08/learningguides/GU0122%

5C%20C-to-Hardware%5C%20Compiler%5C%20User%5C%20Manual.pdf.

[9] Altium Ltd. *C-to-Hardware Compiler User Manual*. 19th May 2013. URL: http://www.altium.com/files/altiumdesigner/s08/learningguides/GU0122%5C%20C-to-Hardware%5C%20Compiler%5C%20User%5C%20Manual.pdf.

[10] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: http://doi.acm.org/10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[11] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ed Komp, Ron Sass, and David Andrews. "Enabling a Uniform Programming Model Across the Software/Hardware Boundary." In: *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–98. ISBN: 0-7695-2661-6. DOI: 10.1109/FCCM.2006.40. URL: http://dx.doi.org/10.1109/FCCM.2006.40.

[12] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, Wei Song, J. Mawer, A. Cristal, and M. Lujan. "An empirical evaluation of High-Level Synthesis languages and tools for database acceleration." In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927484.

[13] Arvind, V. Kathail, and K. Pingali. *A Dataflow Architecture with Tagged Tokens*. Technical memoranda. Massachusetts Inst. of Technology, Laboratory for Computer Science, 1980. URL: https://books.google.de/books?id=5wosGwAACAAJ.

[14] J. L. Bass. *FPGA C Compiler*. 24th Feb. 2011. URL: http://sourceforge.net/projects/fpgac.

[15] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. "PACT XPP—A Self-Reconfigurable Data Processing Architecture." In: *The Journal of Super-*

computing 26.2 (2003), pp. 167–184. DOI: 10.1023/a:1024499601571.

[16] Y. Ben Asher and N. Rotem. "Binary Synthesis with multiple memory banks targeting array references." In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. Aug. 2009, pp. 600–603. DOI: 10.1109/FPL.2009.5272381.

[17] Y. Ben-Asher and N. Rotem. "Synthesis for variable pipelined function units." In: *System-on-Chip, 2008. SOC 2008. International Symposium on*. Nov. 2008, pp. 1–4. DOI: 10.1109/ISSOC.2008.4694874.

[18] Yosi Ben-Asher, Danny Meisler, and Nadav Rotem. "Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs." In: *ACM Trans. Reconfigurable Technol. Syst.* 3.3 (Sept. 2010), 15:1–15:19. ISSN: 1936-7406. DOI: 10.1145/1839480.1839485. URL: http://doi.acm.org/10.1145/1839480.1839485.

[19] Yosi Ben-Asher and Nadav Rotem. "Automatic Memory Partitioning: Increasing Memory Parallelism via Data Structure Partitioning." In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*. CODES/ISSS '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 155–162. ISBN: 978-1-60558-905-3. DOI: 10.1145/1878961.1878989. URL: http://doi.acm.org/10.1145/1878961.1878989.

[20] Koen Bertels et al. "The hArtes Tool Chain." In: *Hardware/Software Co-design for Heterogeneous Multi-core Platforms: The hArtes Toolchain*. Ed. by Koen Bertels. Dordrecht: Springer Netherlands, 2012, pp. 9–109. ISBN: 978-94-007-1406-9. DOI: 10.1007/978-94-007-1406-9_2. URL: https://doi.org/10.1007/978-94-007-1406-9_2.

[21] Nguyen Ngoc Bình, Masaharu Imai, Akichika Shiomi, and Nobuyuki Hikichi. "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts." In: *Proceedings of the 33rd annual Design Automation Conference*. DAC '96. Las Vegas, Nevada, United States: ACM, 1996, pp. 527–532. ISBN: 0-89791-779-0. DOI: http://doi.acm.org/10.1145/240518.240618. URL: http://doi.acm.org/10.1145/240518.240618.

[22] Moritz Blauert, Reimond Retz, Daniel Schäfer, Jörg Schmalfuß, and Juha Ojansivu. "Bachelorpraktikum – Graphenvisualisierung VeriDeBug." Comp. software. Technische Universität Darmstadt, 2009.

[23] C. Böhm and G. Jacopini. "Flow diagrams, turing machines and languages with only two formation rules." In: *Communications of the ACM* 9.5 (1966), pp. 366–371.

[24] M. Bowen and Embedded Solutions Ltd. *Handel-C Language Reference Manual*. Version 2.1. 13th Jan. 2009. URL: http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%5C%20C.PDF.

[25] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. "Practical Improvements to the Construction and Destruction of Static Single Assignment Form." In: *Softw. Pract. Exper.* 28.8 (July 1998), pp. 859–881. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8. URL: http://dx.doi.org/10.1002/(SICI)1097-024X(19980710)28:8%3C859::AID-SPE188%3E3.0.CO;2-8.

[26] M. Budiu. "Spatial Computation." PhD thesis. Carnegie Mellon University, Pittsburgh, USA: School of Computer Science, Carnegie Mellon University, Dec. 2003.

[27] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Michael Dahlin, Lizy Kurian John, Calvin Lin, Charles R. Moore, James H. Burrill, Robert G. McDonald, and William Yode. "Scaling to the End of Silicon with EDGE Architectures." In: *IEEE Computer* 37.7 (2004), pp. 44–55. URL: http://dblp.uni-trier.de/db/journals/computer/computer37.html#BurgerKMDJLMBMY04.

[28] B. Buyukkurt and W.A. Najjar. "Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs." In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Heidelberg, Germany, Sept. 2008, pp. 655–658.

[29] C. P. Steffen. "Nallatech Software Tools on the Maxwell FPGA Parallel Computer." In: *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*. Urbana, Illinois, USA, July 2008.

[30] *C2R Compiler DataSheet*. URL: http://www.altera.com/literature/ug/ug%20nios2%20c2h%20compiler.pdf.

[31] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. "The Garp architecture and C compiler." In: *IEEE Computer* 33.4 (Apr. 2000), pp. 62–69.

[32] Timothy J. Callahan and John Wawrzynek. "Adapting Software Pipelining for Reconfigurable Computing." In: *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '00. San Jose, California, USA: ACM, 2000, pp. 57–64. ISBN: 1-58113-338-3. DOI: 10.1145/354880.354889. URL: http://doi.acm.org/10.1145/354880.354889.

[33] Timothy John Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. Tech. rep. UNIVERSITY OF CALIFORNIA BERKELEY, 2002.

[34] Calypto Design Systems, Inc. *Catapult Product Family Datasheet*. 2014. URL: http://calypto.com/en/page/leadform/31.

[35] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems." In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. URL: http://doi.acm.org/10.1145/1950413.1950423.

[36] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems." In: *ACM Trans. Embed. Comput. Syst.* 13.2 (Sept. 2013), 24:1–24:27. ISSN: 1539-9087. DOI: 10.1145/2514740. URL: http://doi.acm.org/10.1145/2514740.

[37] Andrew Canis, Jongsok Choi, Blair Fort, Bain Syrowik, Ruo Long Lian, Yu Ting Chen, Hsuan Hsiao, Jeffrey Goeders, Stephen Brown, and Jason Anderson. "LegUp High-Level Synthesis." In: *FPGAs for Software Programmers*. Ed. by Dirk Koch, Frank Hannig, and Daniel Ziener. Cham: Springer International Publishing, 2016, pp. 175–190. ISBN: 978-3-319-26408-0. DOI: 10.1007/978-3-319-

26408-0_10. URL: https://doi.org/10.1007/978-3-319-26408-0_10.

[38] João M.P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer US, 2009. DOI: 10.1007/978-0-387-09671-1.

[39] João M.P. Cardoso and Markus Weinhardt. "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture." In: *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream: 12th International Conference, FPL 2002 Montpellier, France, September 2–4, 2002 Proceedings*. Ed. by Manfred Glesner, Peter Zipf, and Michel Renovell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 864–874. ISBN: 978-3-540-46117-3. DOI: 10.1007/3-540-46117-5_89. URL: https://doi.org/10.1007/3-540-46117-5_89.

[40] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms." In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: ACM, 2016, 109:1–109:6. ISBN: 978-1-4503-4236-0. DOI: 10.1145/2897937.2897972. URL: http://doi.acm.org/10.1145/2897937.2897972.

[41] COINS Homepage. *COINS: A Compiler Infrastructure*. 24th Feb. 2014. URL: http://www.coins-project.org.

[42] Convey Computer. *Convey HC-1 Computer – Architecture Overiew*. Convey Computer Corporation. Nov. 2008. URL: http://www.conveycomputer.com/Resources/ConveyArchitectureWhiteP.pdf.

[43] Convey Computer. *HC Family Brochure*. Convey Computer Corporation. 2010. URL: http://www.conveycomputer.com/Resources/Convey_HC1_Family.pdf.

[44] J. Cong, Yiping Fan, G. Han, Wei Jiang, and Zhiru Zhang. "Platform-Based Behavior-Level and System-Level Synthesis." In: *SOC Conference, 2006 IEEE International*. Sept. 2006, pp. 199–202. DOI: 10.1109/SOCC.2006.283880.

[45] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491.

[46] Altera Corporation. *EPXA1 Development Board Hardware Reference Manual*. 2002.

[47] Altera Corporation. *SRC Selects Altera for Next Generation MAP Processor*. 2007. URL: https://www.prnewswire.com/news-releases/src-selects-altera-for-next-generation-map-processor-59836522.html.

[48] Altera Corporation. *Arria V Device Handbook*. Nov. 2011. URL: http://www.altera.com/literature/hb/arria-v/arriav_handbook.pdf.

[49] Altera Corporation. *Cyclone V Device Handbook*. Nov. 2011. URL: http://www.altera.com/literature/hb/cyclone-v/cyclone5_handbook.pdf.

[50] NVIDIA Corporation. *NVIDIA TESLA V100 GPU ARCHITECTURE*. NVIDIA Corporation. 2017.

[51] CriticalBlue. *Cascade Datasheet*. CriticalBlue Limited. 2006.

[52] CriticalBlue. *Cascade Product Overview*. CriticalBlue, Inc. 2007.

[53] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (Oct. 1991), pp. 451–490.

[54] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. "From opencl to high-performance hardware on FPGAS." In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2012, pp. 531–534. DOI: 10.1109/FPL.2012.6339272.

[55] Luka Daoud, Dawid Zydek, and Henry Selvaraj. "A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing." English. In: *Advances in Systems Science*. Ed. by Jerzy Swiątek, Adam Grzech, Paweł Swiątek, and Jakub M. Tomczak. Vol. 240. Advances in Intelligent Systems and Computing. Springer International Publishing, 2014, pp. 483–492. ISBN: 978-3-319-01856-0. DOI: 10.1007/978-3-319-01857-7_47. URL: http://dx.doi.org/10.1007/978-3-319-01857-7_47.

[56] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, eds. *Readings in Hardware/Software Co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002. ISBN: 1-55860-702-1.

[57] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-flow Processor." In: *SIGARCH Comput. Archit. News* 3.4 (Dec. 1974), pp. 126–132. ISSN: 0163-5964. DOI: 10.1145/641675.642111. URL: http://doi.acm.org/10.1145/641675.642111.

[58] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-flow Processor." In: *Proceedings of the 2Nd Annual Symposium on Computer Architecture*. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. DOI: 10.1145/642089.642111. URL: http://doi.acm.org/10.1145/642089.642111.

[59] Robert P. Dick and Niraj K. Jha. "MOGAC: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems." In: *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. ICCAD '97. San Jose, California, United States: IEEE Computer Society, 1997, pp. 522–529. ISBN: 0-8186-8200-0. URL: http://dl.acm.org/citation.cfm?id=266388.266544.

[60] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. "The active database management system manifesto: A rulebase of ADBMS features." In: *Rules in Database Systems*. Ed. by Timos Sellis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–17. ISBN: 978-3-540-45137-2.

[61] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search." In: *Design Automation for Embedded Systems* 2.1 (1997), pp. 5–32. URL: http://www.springerlink.com/index/N0M725Q46126V741.pdf.

[62] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. "Graphviz — open source graph drawing tools." In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 483–484.

[63] Rolf Ernst, Jorg Henkel, and Thomas Benner. "Hardware-Software Cosynthesis for Microcontrollers." In: *IEEE Des. Test* 10.4 (Oct. 1993), pp. 64–75. ISSN: 0740-7475. DOI: 10.1109/54.245964. URL: http://dx.doi.org/10.1109/54.245964.

[64] A. M. Erosa and L. J. Hendren. "Taming control flow: a structured approach to eliminating goto statements." In: *Computer Languages, 1994., Proceedings of the 1994 International Conference on*. May 1994, pp. 229–240. DOI: 10.1109/ICCL.1994.288377.

[65] Jan Frigo, Maya Gokhale, and Dominique Lavenier. "Evaluation of the streams-C C-to-FPGA Compiler: An Applications Perspective." In: *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. FPGA '01. Monterey, California, USA: ACM, 2001, pp. 134–140. ISBN: 1-58113-341-3. DOI: 10.1145/360276.360326. URL: http://doi.acm.org/10.1145/360276.360326.

[66] Hagen Gädke-Lütjens, Benjamin Thielmann, and Andreas Koch. "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation." In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2010.

[67] Hagen Gädke. "Dynamic Scheduling in High-Level Compilation for Adaptive Computers." PhD thesis. Technische Universit"at Braunschweig, 2011.

[68] Hagen Gädke, Florian Stock, and Andreas Koch. "Memory access parallelisation in high-level language compilation for reconfigurable adaptive computers." In: *FPL*. IEEE, 2008, pp. 403–408.

[69] D. Gajski and M. Reshadi. "A cycle-accurate compilation algorithm for custom pipelined datapaths." In: *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*. Sept. 2005, pp. 21–26. DOI: 10.1145/1084834.1084845.

[70] Bertrand Le Gal, Emmanuel Casseau, Sylvain Huet, Pierre Bomel, Christophe Jego, and Eric Martin. "C-based Rapid Prototyping For Digital Signal Processing." In: *Proceedings of the 13th European Signal Processing Conference (EUSIPCO)*. Antalya, Turkey, Sept. 2005.

[71] D. Galloway. "The Transmogrifier C hardware description language and compiler for FPGAs." In: *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*. Apr. 1995, pp. 136–144. DOI: 10.1109/FPGA.1995.477419.

[72] Dave Galloway. *Transmogrifier C*. University of Toronto. 1997. URL: http://www.eecg.toronto.edu/~jayar/research/tmcc.ps.

[73] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some Simplified NP-complete Problems." In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*. STOC '74. Seattle, Washington, USA: ACM, 1974, pp. 47–63. DOI: 10.1145/800119.803884. URL: http://doi.acm.org/10.1145/800119.803884.

[74] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[75] G. Genest, R. Chamberlain, and R. Bruce. "Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C." In: *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. Aug. 2007, pp. 280–286. DOI: 10.1109/AHS.2007.89.

[76] M. B. Gokhale and J. M. Stone. "NAPA C: compiling for a hybrid RISC/FPGA architecture." In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. Apr. 1998, pp. 126–135. DOI: 10.1109/FPGA.1998.707890.

[77] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. "Stream-Oriented FPGA Computing in the Streams-C High Level Language." In: *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 49–. ISBN: 0-7695-0871-5. URL: http://dl.acm.org/citation.cfm?id=795659.795916.

[78] Maya Gokhale, James Kaba, and Aaron Marks andJang Kim. *Malleable architecture generator for FPGA computing*. 1996. DOI: 10.1117/12.255818. URL: http://dx.doi.org/10.1117/12.255818.

[79] Zhi Guo, Walid Najjar, and Betul Buyukkurt. "Efficient Hardware Code Generation for FPGAs." In: *ACM Transactions on Architectures and Code Optimization (TACO)* 5.1 (May 2008), pp. 1–26.

[80] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations." In: *Proceedings of the 16th International Conference on VLSI Design (VLSI)*. New Delhi, India, Jan. 2003, pp. 461–466.

[81] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Norwell, Massachusetts, USA: Kluwer Academic Publishers, 2004.

[82] J. R. Gurd. "The Manchester dataflow machine." In: *Future Generation Computer Systems* 1.4 (1985), pp. 201–212.

[83] S. Hadjis, A. Canis, R. Sobue, Y. Hara-Azumi, H. Tomiyama, and J. Anderson. "Profiling-driven multicycling in FPGA high-level synthesis." In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2015, pp. 31–36. DOI: 10.7873/DATE.2015.0512.

[84] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, Shih-Wei Liao, E. Bugnion, and M. S. Lam. "Maximizing multiprocessor performance with the SUIF compiler." In: *Computer* 29.12 (Dec. 1996), pp. 84–89. ISSN: 0018-9162. DOI: 10.1109/2.546613.

[85] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. "CHStone: A benchmark program suite for practical C-based high-level synthesis." In: *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. May 2008, pp. 1192–1195. DOI: 10.1109/ISCAS.2008.4541637.

[86] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis." In: *Journal of Information Processing*. Vol. 17. 2009, pp. 242–254.

[87] J. R. Hauser and J. Wawrzynek. "Garp: a MIPS processor with a reconfigurable coprocessor." In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 1997.

[88] Jörg Henkel, Rolf Ernst, Ullrich Holtmann, and Thomas Benner. "Adaptation of Partitioning and High-level Synthesis in Hardware/Software Co-synthesis." In: *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '94. San Jose, California, USA: IEEE Computer Society Press, 1994, pp. 96–100. ISBN: 0-89791-690-5. URL: http://dl.acm.org/citation.cfm?id=191326.175193.

[89] Dietmar Hildenbrand, Holger Lange, Florian Stock, and Andreas Koch. "Efficient Inverse Kinematics Algorithm Based on Conformal Geometric Algebra - Using Reconfigurable Hardware." In: *GRAPP*. Ed. by Alpesh Ranchordas and Helder Araújo. INSTICC - Institute for Systems, Technologies of Information, Control, and Communication, 2008, pp. 300–307. ISBN: 978-989-8111-21-0.

[90] C. A. R. Hoare. "Communicating Sequential Processes." In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[91] Giang Nguyen Thi Huong and Seon Wook Kim. "GCC2Verilog Compiler Toolset for Complete Translation of C Programming Language into Verilog HDL." In: *ETRI Journal* 33.5 (Oct. 2011), pp. 731–740. DOI: http://dx.doi.org/10.4218/etrij.11.0110.0654.

[92] Huppenthal. *SRC Architecture – Implicit + Explicit*. Tech. rep. SRC Computers, Inc., 2005.

[93] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. "Hardware/software co-compilation with the Nymble system." In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581538.

[94] Jens Huthmann. "Hardware-Software-Partitioning using the Scale-Compiler-Framework." Diplomthesis. TU Darmstadt, 2009.

[95] Jens Christoph Huthmann. "An Execution Model and High-Level-Synthesis System for Generating SIMT Multi-Threaded Hardware from C Source Code." PhD thesis. Darmstadt: Technische Universität, 2017. URL: http://tuprints.ulb.tu-darmstadt.de/6776/.

[96] Jens Huthmann, Peter Muller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Accelerating high-level engineering computations by automatic compilation of Geometric Algebra to hardware accelerators." In: *ICSAMOS*. Ed. by Fadi J. Kurdahi and Jarmo Takala. IEEE, 2010, pp. 216–222. ISBN: 978-1-4244-7937-5.

[97] Jens Huthmann, Peter Müller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Accelerating High-Level Engineering Computations by Automatic Compilation of Geometric Algebra to Hardware Accelerators." In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation*. Samos, Greece, July 2010.

[98] Jens Huthmann, Peter Müller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators." In: *Dynamically Reconfigurable Architectures*. Ed. by Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid Verbauwhede. Dagstuhl Seminar Proceedings 10281. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2838.

[99] Impulse Accelerated Technologies Inc. *Impulse-C Website*. Feb. 2011. URL: http://www.impulseaccelerated.com.

[100] Cray Inc. *Cray XD1 Datasheet*. Cray Inc. Seattle, Washington, USA, 2004.

[101] National Instruments. *LabVIEW 2014 FPGA Module Help*. National Instruments. 2014. URL: http://www.ni.com/pdf/manuals/371599k.zip.

[102] Intel. *Intel FPGA SDK for OpenCL Getting Started Guide*. Intel Corporation. 2016. URL: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf.

[103] Intel. *Intel FPGA SDK for OpenCL Best Practices Guide*. Intel Corporation. 2017. URL: https://www.altera.com/documentation/mwh1391807516407.htm.

[104] Y. Ito, Y. Sugawara, M. Inaba, and K. Hiraki. "CVC: The C to RTL compiler for callback-based verification model." In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. Sept. 2008, pp. 499–502. DOI: 10.1109/FPL.2008.4629993.

[105] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming." In: *IFIP Congress*. 1974, pp. 471–475. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Kahn74.

[106] R. Karp and R. Miller. "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing." In: *SIAM Journal on Applied Mathematics* 14.6 (1966), pp. 1390–1411. DOI: 10.1137/0114108. eprint: http://dx.doi.org/10.1137/0114108. URL: http://dx.doi.org/10.1137/0114108.

[107] Nico Kasprzyk. "COMRADE - Ein Hochsprachen-Compiler für adaptive Computersysteme." PhD thesis. Mühlenpfordtstr. 23, 38106 Braunschweig: Abteilung Entwurf integrierter Schaltungen, Technische Universität Braunschweig, June 2005.

[108] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D.C. Cronquist, and M. Sivaraman. "PICO: automatically designing custom computers." In: *Computer* 35.9 (Sept. 2002), pp. 39–47. ISSN: 0018-9162. DOI: 10.1109/MC.2002.1033026.

[109] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. "The Omega Calculator and Library, Version 1.1.0." In: (1996). URL: http://www.cs.utah.edu/~mhall/cs6963s09/lectures/omega.ps.

[110] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[111] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988. ISBN: 0-13-110370-9.

[112] Volodymyr V. Kindratenko. "Code Partitioning for Reconfigurable High-Performance Computing: A Case Study." In: *The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Ed. by Toomas P. Plaks. CSREA Press, Dec. 11, 2006, pp. 143–152. ISBN: 1-60132-011-6. URL: http://dblp.uni-trier.de/db/conf/ersa/ersa2006.html#Kindratenko06.

[113] Volodymyr V. Kindratenko, Adam D. Myers, and Robert J. Brunner. "Implementation of the Two-point Angular Correlation Function on a High-performance Reconfigurable Computer." In: *Sci. Program.* 17.3 (Aug. 2009), pp. 247–259. ISSN: 1058-9244. DOI: 10.1155/2009/434735. URL: http://dx.doi.org/10.1155/2009/434735.

[114] Andreas Koch. *Advances in Adaptive Computer Technology*. Habilitation, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, Germany. Dec. 2004.

[115] Andreas Koch and Nico Kasprzyk. "High-Level-Language Compilation for Reconfigurable Computers." In: *Proceedings of the International Conference on Reconfigurable Communication-centric SoCs (ReCoSoC)*. Montpellier, France, June 2005.

[116] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines." In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328. ISSN: 0362-1340. DOI: 10.1145/960116.54022. URL: http://doi.acm.org/10.1145/960116.54022.

[117] Holger Lange and Andreas Koch. "Memory Access Schemes for Configurable Processors." In: *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL)*. Villach, Austria: Springer-Verlag, Aug. 2000, pp. 615–625.

[118] Holger Lange and Andreas Koch. "Low-latency high-bandwidth HW/SW communication in a virtual memory environment." In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Heidelberg, Germany, Sept. 2008, pp. 281–286.

[119] Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. "Acceleration and Energy Efficiency of a Geometric Algebra Computation using Reconfigurable Computers and GPUs." In: *FCCM*. Ed. by Kenneth L. Pocek and Duncan A. Buell. IEEE Computer Society, 2009, pp. 255–258. ISBN: 978-0-7695-3716-0.

[120] M. Langhammer. "Floating point datapath synthesis for FPGAs." In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. Sept. 2008, pp. 355–360. DOI: 10.1109/FPL.2008.4629963.

[121] C. Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization." MA thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[122] David Lau, Orion Pritchard, and Philippe Molson. "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions." In: *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 45–56.

[123] S. Lee, D. Raila, and V. Kindratenko. "LLVM-CHiMPS: compilation environment for FPGAs using LLVM compiler infrastructure and CHiMPS computational model." In: *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*. Urbana, Illinois, USA, July 2008.

[124] D.M. Lewis, D.R. Galloway, M. van Ierssel, J. Rose, and P. Chow. "The Transmogrifier-2: a 1 million gate rapid-prototyping system." In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 6.2 (June 1998), pp. 188–198. ISSN: 1063-8210. DOI: 10.1109/92.678867.

[125] D. MacMillen. *Nimble Compiler Environment for Agile Hardware*. Storming Media LLC (USA), 2001.

[126] Jan Madsen, Jesper Grode, Peter Voigt Knudsen, M. E. Petersen, and Anne Elisabeth Haxthausen. "LYCOS: the Lyngby Co-Synthesis System." In: *Design Autom. for Emb. Sys.* 2.2 (1997), pp. 195–235.

[127] Zoltán Ádám Mann, András Orbán, and Péter Arató. "Finding optimal hardware/software partitions." In: *Form. Methods Syst. Des.* 31 (3 Dec. 2007), pp. 241–263. ISSN: 0925-9856. DOI: 10.1007/s10703-007-0039-0. URL: http://dl.acm.org/citation.cfm?id=1315662.1315669.

[128] ZoltáN Mann, AndráS OrbáN, and Viktor Farkas. "Evaluating the Kernighan-Lin Heuristic for Hardware/Software Partitioning." In: *Int. J. Appl. Math. Comput. Sci.* 17 (2 June 2007), pp. 249–267. ISSN: 1641-876X. DOI: http://dx.doi.org/10.2478/v10006-007-0022-3. URL: http://dx.doi.org/10.2478/v10006-007-0022-3.

[129] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future." In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 18–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.83.

[130] The MathWorks. *HDL Coder Reference*. The MathWorks. 2015. URL: http://www.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ref.pdf.

[131] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. "An overview of today's high-level synthesis tools." In: *Design Automation for Embedded Systems* 16.3 (Sept. 2012), pp. 31–51. ISSN: 1572-8080. DOI: 10.1007/s10617-012-9096-8. URL: https://doi.org/10.1007/s10617-012-9096-8.

[132] Mentor Graphics Corp. *Catapult-C data sheet*. 15th Jan. 2009. URL: http://www.mentor.com/products/esl/high%5C_level%5C_synthesis/catapult%5C_synthesis/upload/Catapult%5C_DS%5C_pdf.pdf.

[133] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[134] Mitrionics. *Mitrion Users' Guide*. 24th Feb. 2011. URL: http://forum.mitrionics.com/uploads/Mitrion%5C_Users%5C_Guide.pdf.

[135] Gordon E Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE SolidState Circuits Newsletter* 20.3 (2006), pp. 33–35. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4785860.

[136] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[137] Rene Mueller, Jens Teubner, and Gustavo Alonso. "Streams on wires: a query compiler for FPGAs." In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 229–240. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687627.1687654.

[138] T. Murata. "Petri nets: Properties, analysis and applications." In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580.

[139] NEC. *CYBERWORKBENCH – NEC's High Level Synthesis Solution*. 2016.

[140] T. Neumann and A. Koch. "A Generic Library for Adaptive Computing Environments." In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Vol. 2147/2001. 2001, pp. 503–512.

[141] Ralf Niemann. *Hardware/Software CO-Design for Data Flow Dominated Embedded Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792382994.

[142] Ralf Niemann and Peter Marwedel. "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming." In: *Proceedings of the ED&TC*. Kluwer Academic Publishers, 1996, pp. 165–193.

[143] H. Nikolov, T. Stefanov, and E. Deprettere. "Systematic and Automated Multiprocessor System Design, Programming, and Implementation." In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.3 (Mar. 2008), pp. 542–555. ISSN: 0278-0070. DOI: 10.1109/TCAD.2007.911337.

[144] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. "Daedalus: Toward Composable Multimedia MP-SoC Design." In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 574–579. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391615. URL: http://doi.acm.org/10.1145/1391469.1391615.

[145] Elena Moscu Panainte. "The Molen Compiler for Reconfigurable Architectures." PhD thesis. Technical University Delft, June 2007.

[146] Y.V. Panchul, D.A. Soderman, and D.R. Coleman. *System for converting hardware designs in high-level programming language to hardware implementations*. US Patent 6,226,776. May 2001. URL: https://www.google.de/patents/US6226776.

[147] T. J. Parr and R. W. Quong. "ANTLR: A predicated-LL(k) parser generator." In: *Softw: Pract. Exper.* 25.7 (July 1995), pp. 789–810. ISSN: 0038-0644. DOI: 10.1002/spe.4380250705. URL: http://dx.doi.org/10.1002/spe.4380250705.

[148] Latinka Pavlova, Nedislav Nedyalkov, Nikola Dyundev, and Henrik Schröder. "Bachelorpraktikum – Graphenvisualisierung pnnsGraph." Comp. software. Technische Universität Darmstadt, 2009.

[149] Colin Percival. *STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS*. Ottawa, Canada, 2009. URL: http://www.tarsnap.com/scrypt/scrypt.pdf.

[150] Carl Adam Petri. "Kommunikation mit Automaten." ger. PhD thesis. Universität Hamburg, 1962.

[151] *PhoenixProject CMU Reconfigurable Nanotechnology*. 2007. URL: https://www.cs.cmu.edu/~phoenix/compiler.html (visited on 2017).

[152] Christian Pilato and Fabrizio Ferrandi. "Bambu: A Free Framework for the High Level Synthesis of Complex Applications." In: *DATE Conference Poster 2012*. 2010.

[153] Bryan T. Preas and Michael Lorenzetti. *Physical Design Automation of Vlsi Systems*. Benjamin-Cummings Pub Co, 1988.

[154] William Pugh. "Uniform Techniques for Loop Optimization." In: *5th International Conference on Supercomputing (ICS'91)*. ACM. 1991, pp. 341–352.

[155] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures." In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Heidelberg, Germany, Sept. 2008, pp. 173–178.

[156] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. "Performance and power of cache-based reconfigurable computing." In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Monterey, California, USA: ACM, 2009, pp. 281–281.

[157] R. Razdan and M. D. Smith. "A High-Performance Microarchitecture with Hardware-Programmable Functional Units." In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. San Jose, CA, USA: ACM New York, NY, USA, Dec. 1994, pp. 172–180.

[158] M. Reshadi, B. Gorjara, and D. Gajski. "C-based design flow: A case study on G.729A for Voice over internet protocol (VoIP)." In: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. June 2008, pp. 72–75.

[159] D. L. Rosenband and Arvind. "Hardware synthesis from guarded atomic actions with performance specifications." In: *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* Nov. 2005, pp. 784–791. DOI: 10.1109/ICCAD.2005.1560170.

[160] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. "The NAPA adaptive processing architecture." In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. Apr. 1998, pp. 28–37. DOI: 10.1109/FPGA.1998.707878.

[161] R. Sakurai, M. Takahashi, A. Kay, A. Yamada, T. Fujimoto, and T. Kambe. "A scheduling method for synchronous communication in the Bach hardware compiler." In: *Proceedings of the Asian and South Pacific Design Automation Conference (ASP-DAC)*. Vol. 1. Jan. 1999, pp. 193–196.

[162] Masataka Sass, Toshiharu Nakaya, Masaki Koham, Takeaki Fukuoka, Masahito Takahashiand, and Ikuo Nakata. "Static Single Assignment Form in the COINS Compiler Infrastructure - Current Status and Background." In: *Proc. of JSSST Workshop on Programming and Application Systems (SPA2003)* (2003).

[163] Scale Compiler Group. *Scale Compiler Homepage*. 24th Feb. 2011. URL: http://www.cs.utexas.edu/users/cart/Scale/.

[164] Christian Schwinn, Dietmar Hildenbrand, Florian Stock, and Andreas Koch. "Gaalop 2.0 - A Geometric Algebra Algorithm Compiler." In: GraVisMa 2010 Proceedings (2010). Ed. by Václav Skala and Eckhard M.

[165] Marc Shapiro and Susan Horwitz. "Fast and accurate flow-insensitive points-to analysis." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Paris, France: ACM, 1997, pp. 1–14.

[166] Richard Sharp. *Higher-Level Hardware Synthesis*. Springer Berlin Heidelberg, 2004. DOI: 10.1007/b95732.

[167] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinle, and Jim Burrill. "Compiling for EDGE Architectures." In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 185–195. ISBN: 0-7695-2499-0. DOI: 10.1109/CGO.2006.10. URL: http://dx.doi.org/10.1109/CGO.2006.10.

[168] M. D. Smith and G. Holloway. *An Introduction to machine SUIF and its Portable Libraries for Analysis and Optimization*. Tech. rep. Cambridge, MA, USA: Division of Engineering and Applied Sciences, Harvard University, 2002.

[169] TIOBE SOFTWARE. *TIOBE Programming Community Index*. Accessed 2012-01-30. 2012. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[170] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. ISBN: 144141276X, 9781441412768.

[171] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time." In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41.

[172] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. "System design using Khan process networks: the Compaan/Laura approach." In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. Vol. 1. Feb. 2004, 340–345 Vol.1. DOI: 10.1109/DATE.2004.1268870.

[173] Luca Sterpone. "Radiation Effects on SRAM-Based FPGAs." In: *Electronics System Design Techniques for Safety Critical Applications*. Dordrecht: Springer Netherlands, 2009, pp. 17–45. ISBN: 978-1-4020-8979-4. DOI: 10.1007/978-1-4020-8979-4_2. URL: https://doi.org/10.1007/978-1-4020-8979-4_2.

[174] F. Stock, A. Koch, and D. Hildenbrand. "FPGA-accelerated color edge detection using a Geometric-Algebra-to-Verilog compiler." In: (Oct. 2013), pp. 1–6. DOI: 10.1109/ISSoC.2013.6675272.

[175] Florian Stock. "Development of Parameterized Placement and Routing Software Tools for Coarse-Granular Reconfigurable Devices." Diplomarbeit. Technische Universit"at Braunschweig, 2005.

[176] Florian Stock and Andreas Koch. "Architecture Exploration and Tools for Pipelined Coarse-Grained Reconfigurable Arrays." In: *FPL*. IEEE, 2006, pp. 1–6.

[177] Florian Stock and Andreas Koch. "A Fast GPU Implementation for Solving Sparse Ill-Posed Linear Equation Systems." In: *PPAM (1)*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski. Vol. 6067. Lecture Notes in Computer Science. Springer, 2009, pp. 457–466. ISBN: 978-3-642-14389-2.

[178] Mizuki Takahashi, Nagisa Ishiura, Akihisa Yamada, and Takashi Kambe. "Thread Partitioning Method for Hardware Compiler Bach." In: *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*. ASP-DAC '00. Yokohama, Japan: ACM, 2000, pp. 303–308. ISBN: 0-7803-5974-7. DOI: 10.1145/368434.368647. URL: http://doi.acm.org/10.1145/368434.368647.

[179] Dimitris Theodoropoulos, Yana Yankova, Georgi Kuzmanov, and Koen Bertels. "Automatic hardware generation for the Molen reconfigurable architecture: a G721 case study." In: (Nov. 2017).

[180] Benjamin Thielmann and Hagen Gädke-Lütjens. *Modlib: A Flexible Module Library for High-Level Language Compilation*. Tech. rep. Darmstadt, Germany: FG Embedded Systems and Applications, TU Darmstadt, Aug. 2010.

[181] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. "Precore - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation." In: *International Conference on Field Programmable Logic and Applications, FPL 2011, September 5-7, Chania, Crete, Greece*. IEEE Computer Society, 2011, pp. 123–129. ISBN: 978-1-4577-1484-9. DOI: 10.1109/FPL.2011.31. URL: https://doi.org/10.1109/FPL.2011.31.

[182] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MPSoCs." In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '07. Salzburg, Austria: ACM, 2007, pp. 9–14. ISBN: 978-1-59593-824-4. DOI: 10.1145/1289816.1289823. URL: http://doi.acm.org/10.1145/1289816.1289823.

[183] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale. "Trident: an FPGA compiler framework for floating-point algorithms." In: *International Conference on Field Programmable Logic and Applications, 2005.* Aug. 2005, pp. 317–322. DOI: 10.1109/FPL.2005.1515741.

[184] Justin L. Tripp, Preston A. Jackson, and Brad Hutchings. "Sea Cucumber: A Synthesizing Compiler for FPGAs." In: *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*. FPL '02. London, UK, UK: Springer-Verlag, 2002, pp. 875–885. ISBN: 3-540-44108-5. URL: http://dl.acm.org/citation.cfm?id=647929.740232.

[185] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software (ICMS'10)*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. LNCS 6327. Springer-Verlag, 2010, pp. 299–302.

[186] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. "pn: A Tool for Improved Derivation of Process Networks." In: *EURASIP Journal on Embedded Systems* 2007.1 (2007), p. 075947. ISSN: 1687-3963. DOI: 10.1155/2007/75947. URL: http://jes.eurasipjournals.com/content/2007/1/075947.

[187] Jason R. Villarreal, Adrian Park, Walid A. Najjar, and Robert Halstead. "Designing Modular Hardware Accelerators in C with ROCCC 2.0." In: *FCCM*. Ed. by Ron Sass and Russell Tessier. IEEE Computer Society, 2010, pp. 127–134. ISBN: 978-0-7695-4056-6.

[188] K. Vipin and S.A. Fahmy. "Automated Partitioning for Partial Reconfiguration Design of Adaptive Systems." In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. May 2013, pp. 172–181. DOI: 10.1109/IPDPSW.2013.119.

[189] Kazutoshi Wakabayashi. "C-based synthesis experiences with a behavior synthesizer, "cyber"." In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. Munich, Germany: ACM, Mar. 1999, pp. 390–393.

[190] David N. Welton. *Programming Language Popularity*. Accessed 2012-01-30. 2012. URL: http://www.langpop.com.

[191] Niklaus Wirth. "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Commun. ACM* 20.11 (Nov. 1977), pp. 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883. URL: http://doi.acm.org/10.1145/359863.359883.

[192] Florian Wörsdörfer, Florian Stock, Eduardo Bayro-Corrochano, and Dietmar Hildenbrand. "Optimizations and Performance of a Robotics Grasping Algorithm Described in Geometric Algebra." In: *CIARP*. Ed. by Eduardo Bayro-Corrochano and Jan-Olof Eklundh. Vol. 5856. Lecture Notes in Computer Science. Springer, 2009, pp. 263–271. ISBN: 978-3-642-10267-7.

[193] Xilinx. *JBits 2.8*. Xilinx Corporation. Sept. 2000.

[194] Xilinx. *JBits 2.8, Virtex Architecture Guide*. Xilinx Corporation. Sept. 2000.

[195] Xilinx. *JBits 2.8, Virtex Architecture Guide, Slice Internals*. Xilinx Corporation. Sept. 2000.

[196] Xilinx. *Virtex 5 Family Overview (v5.0)*. First Release 2006. Xilinx. Feb. 2009. URL: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

[197] Xilinx. *Virtex 4 Family Overview (v3.1)*. First Release 2004. Xilinx. 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

[198] Xilinx. *Zynq-7000 Extensible Processing Platform Summary*. Xilinx. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[199] Xilinx, Inc. *UltraScale Product Selection Guide*. 30 Nov. 2016. URL: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf.

[200] Xilinx, Inc. *UltraScale Family Overview*. Nov. 2017. URL: http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[201] David Zaretsky, Gaurav Mittal, Xiaoyong Tang, and Prith Banerjee. "Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs." In: *IEEE Symposium on Field-Programmable Custom Computing Machines*. 2004.

[202] Y. Zhu, Y. Liu, D. Zhang, S. Li, P. Zhang, and T. Hadley. "Acceleration of pedestrian detection algorithm on novel C2RTL HW/SW Co-design platform." In: *The 2010 International Conference on Green Circuits and Systems*. June 2010, pp. 615–620. DOI: 10.1109/ICGCS.2010.5542990.

OWN PUBLICATIONS

[68] Hagen Gädke, Florian Stock, and Andreas Koch. "Memory access parallelisation in high-level language compilation for reconfigurable adaptive computers." In: *FPL*. IEEE, 2008, pp. 403–408.

[89] Dietmar Hildenbrand, Holger Lange, Florian Stock, and Andreas Koch. "Efficient Inverse Kinematics Algorithm Based on Conformal Geometric Algebra - Using Reconfigurable Hardware." In: *GRAPP*. Ed. by Alpesh Ranchordas and Helder Araújo. INSTICC - Institute for Systems, Technologies of Information, Control, and Communication, 2008, pp. 300–307. ISBN: 978-989-8111-21-0.

[96] Jens Huthmann, Peter Muller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Accelerating high-level engineering computations by automatic compilation of Geometric Algebra to hardware accelerators." In: *ICSAMOS*. Ed. by Fadi J. Kurdahi and Jarmo Takala. IEEE, 2010, pp. 216–222. ISBN: 978-1-4244-7937-5.

[97] Jens Huthmann, Peter Müller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Accelerating High-Level Engineering Computations by Automatic Compilation of Geometric Algebra to Hardware Accelerators." In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation*. Samos, Greece, July 2010.

[98] Jens Huthmann, Peter Müller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. "Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators." In: *Dynamically Reconfigurable Architectures*. Ed. by Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid Verbauwhede. Dagstuhl Seminar Proceedings 10281. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2838.

[119] Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. "Acceleration and Energy Efficiency of a Geometric Algebra Computation using Reconfigurable Computers and GPUs." In: *FCCM*. Ed. by Kenneth L. Pocek and Duncan A. Buell. IEEE Computer Society, 2009, pp. 255–258. ISBN: 978-0-7695-3716-0.

[164] Christian Schwinn, Dietmar Hildenbrand, Florian Stock, and Andreas Koch. "Gaalop 2.0 - A Geometric Algebra Algorithm Compiler." In: GraVisMa 2010 Proceedings (2010). Ed. by Václav Skala and Eckhard M.

[174] F. Stock, A. Koch, and D. Hildenbrand. "FPGA-accelerated color edge detection using a Geometric-Algebra-to-Verilog compiler." In: (Oct. 2013), pp. 1–6. DOI: 10.1109/ISSoC.2013.6675272.

[175] Florian Stock. "Development of Parameterized Placement and Routing Software Tools for Coarse-Granular Reconfigurable Devices." Diplomarbeit. Technische Universit"at Braunschweig, 2005.

[176] Florian Stock and Andreas Koch. "Architecture Exploration and Tools for Pipelined Coarse-Grained Reconfigurable Arrays." In: *FPL*. IEEE, 2006, pp. 1–6.

[177] Florian Stock and Andreas Koch. "A Fast GPU Implementation for Solving Sparse Ill-Posed Linear Equation Systems." In: *PPAM (1)*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski. Vol. 6067. Lecture Notes in Computer Science. Springer, 2009, pp. 457–466. ISBN: 978-3-642-14389-2.

[192] Florian Wörsdörfer, Florian Stock, Eduardo Bayro-Corrochano, and Dietmar Hildenbrand. "Optimizations and Performance of a Robotics Grasping Algorithm Described in Geometric Algebra." In: *CIARP*. Ed. by Eduardo Bayro-Corrochano and Jan-Olof Eklundh. Vol. 5856. Lecture Notes in Computer Science. Springer, 2009, pp. 263–271. ISBN: 978-3-642-10267-7.

E

ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Quellen verfasst habe.
Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 2018

Dipl.-Inform. Florian Stock

INDEX